Técnicas de aprendizaje estadístico en modelos de valoración dinámica de precios

Roberto Sánchez Latorre

Trabajo de fin de Máster
Universidad de Zaragoza

Directores del trabajo: Tomás Alcalá Nalvaiz y Juan Manuel Soto Juárez
Máster en Modelización e Investigación Matemática, Estadística y Computación
27 de noviembre de 2019

Abstract

The main propose of this work is to study different techniques of reinforcement learning in order to solve some dynamic pricing problems. For that, it is divided in two distinct parts, the theorical chapters and his application to several specific problems. The company *Inycom S.A.* has collaborated by providing a real dynamic pricing problem whose complexity has been reduced for his adaptation to this work.

In the first chapter we introduce dynamic pricing idea and his relationship with demand function. We see the different types of dynamic pricing and a methodology for maximizing sales and so the revenue, for it, the estimation of a demand function which depends directly on price is essential. It is suitable to estimate it as a combination of a deterministic function, $d(p)$, and a random error term, U. At the end of the chapter we present some of the most used demands models.

In the second chapter we study the fundamentals of reinforcement learning for a single agent and we see four algorithms for this type of problems: Q-Learning, SARSA, Double Q-Learning and Expected SARSA. Reinforcement learning is an area of machine learning whose aim is to know how agents should take actions in an environment. In the case of a single agent, it is estructured as a Markov decision process. Markov decision process is an extension of Markov chains and is essentially defined by a state space, an action space and transition probabilities, (S,A,P). This framework provides some useful optimallity statements by the definition of value functions: state-value function and action-value function. The most important results in this chapter are the Bellman equations and in particular, the Bellman optimallity equation because it help us to build algorithms for solving the problem. The different types of algorithms are model-based and free-models, we focus on model free algorithms, in particular the ones who use temporal difference learning.

In the next chapter we study reinforcement learning for multiple agents. For that, we introduce the idea of matrix games and stochastic games. Matrix games are the framework of multi-agent reinforcement learning problems with a single state. In this type of problems we can define the same concepts as in the single agent case, but taking into account that the action space is multidimensional and so the problem is more complex. Stochastic games are its generalization to multiple states, so we have to add transition probabilities and a state space. In these problems the optimal solution for one agent depends directly on the other agents actions so the concept of solution is not clear. In order to deal with this, we can define Nash equilibrium, which guarantee in a certain way an optimal solution for all agents. Finally, we present two main classes of resolution methods: Best-response Learners and Equilibrium Learners.

In last chapter we analyse the results of some dynamic pricing problems solved with reinforcement learning. The first problems have a single agent fixing prices on each month, we have three products with differents demands and a specific inventory level for each product. The goal is to maximize the revenue taking into account the avaliable periods of sales and the storage cost. We compare the results taking a stochastic demand in the training or a deterministic demand. The other type of problems have two agents competing for the sales of the same product. We consider two circumstances, that there is an agent who is not influenced by other's prices and the case that the two agents are mutually influenced by their prices.

Índice general

Capítulo 1

Valoración dinámica de precios

1.1. Introducción

Dynamic pricing o valoración dinámica de precios es una estratégica de fijación de precios que lleva a cabo un vendedor en función de determinadas variables, que pueden ser aspectos internos del negocio como el nivel de stock o aspectos externos como la situación del mercado, la demanda, etc.

El ajuste de estos precios se hace gracias a diversas técnicas que tienen en cuenta estas variables para optimizar el beneficio del negocio. Las técnicas utilizadas pueden ir desde las más lógicas hasta modelos matemáticos muy complejos. Son muy propensos a estas técnicas negocios de venta vía Internet dada la facilidad de recopilar información para estimar las ventas posibles y a su disposición a variar continuamente los precios, un ejemplo de este tipo se puede ver en [15].

Como el concepto y las técnicas de *Dynamic pricing* son muy extensas, vamos a ver distintos tipos de valoración dinámica de precios en función de los elementos del mercado usados y su metodología.

- **Segmentación de precios**. Consiste en ofrecer distintos precios a diferentes productos. Por ejemplo, una estrategia de este tipo sería establecer precios más altos a clientes con mayor poder adquisitivo, ya que están dispuestos a pagar más por un determinado producto.

- **Precios por picos**. Consiste en añadir un valor extra al precio del producto durante un periodo de tiempo que suponga la hora punta de la demanda. Ejemplos de este tipo de estrategias son las usadas por las compañías eléctricas o el precio de los billetes de avión más elevado en vacaciones.

- **Precios de lanzamiento**. Es una estrategia de precios usada para entrar en el mercado. En algunos mercados bastante cerrados es importante ofrecer buenos precios al principio con el fin de fidelizar a los clientes. Una vez ya se tiene una cuota de clientes aceptable, se puede ir subiendo los precios gradualmente para incrementar los beneficios. Un ejemplo de este tipo podrían ser móviles que aparecen en el mercado con un precio bajo pero van aumentando la calidad de sus productos y los precios una vez van captando clientes.

- **Precios basados en la competencia**. Es una estrategia básica que consiste en establecer los precios de venta que tiene la competencia con el fin de mantenerse competitivos en el mercado. Es una técnica muy esencial para la fijación de precios y en cierta manera es usada por todas los negocios.

- **Precios basados en la elasticidad**: El método consiste en intentar estimar la demanda de un determinado producto en función del precio que se fija. A partir de ahí se puede estimar el beneficio esperado de ese producto y maximizarlo.

En la realidad, las técnicas de dynamic pricing no sólo se ajustan a uno de estos tipos, sino que todos estos tipos se presentan mezclados en las metodologías que se usan para fijar precios. Por ejemplo es habitual seguir los precios de la competencia pero analizando tus propias demandas.

El que vamos a desarrollar en el trabajo es el basado en la elasticidad de la demanda ya que se adapta mejor a un análisis matemático del problema, permitiendo analizar el modelo y sacarle el máximo beneficio a la fijación de precios en cada situación. Es un método que necesita del análisis del beneficio esperado, y por lo tanto de la demanda del producto. El desarrollo teórico de este método se puede encontrar en [2].

En particular se tiene que analizar como evoluciona la demanda en función del precio. Para ello es fundamental el concepto de elasticidad de la demanda, que nos permite ver como afecta el precio a los cambios de demanda.

Para desarrollar un buen ajuste de precios es esencial seguir una metodología. En nuestro caso, como nos vamos a fijar en la elasticidad de la demanda, el primer paso se basa en una estimación de la demanda. Si se disponen de datos históricos de ventas y los precios a lo largo del tiempo presentan suficiente variabilidad se puede hacer un análisis estadístico y estimar la demanda en función del precio. Sin embargo, a veces no es fácil conseguir datos de ventas, o el rango de precios históricos no presenta suficiente variabilidad, para estos casos se conocen modelos de demanda muy comunes que pueden ser estudiados. Como este es nuestro caso, en este mismo capítulo veremos varios modelos muy utilizados para estimar la demanda y con ellos realizaremos una valoración dinámica de precios en el capítulo 4. Una vez conocida o estimada la demanda, el siguiente paso es llevar a cabo una optimización del precio para maximizar el beneficio esperado, esta optimización puede no ser sencilla y para realizarla se pueden llevar a cabo técnicas de optimización clásicas, algoritmos genéticos o aprendizaje por refuerzo, como es nuestro caso. El proceso completo de valoración dinámica de precios se verá en el capítulo 4 aplicado a un problema real.

1.2. Modelo de la demanda

Para poder analizar la demanda en profundidad necesitamos construir un modelo matemático que se ajuste a su comportamiento. Consideramos la función de demanda variando en un precio p como la variable aleatoria $X(p)$, a la función de distribución de esta variable la denotamos como $F(x,p)$. La variable aleatoria $X(p)$ se puede dividir en dos partes, una correspondiente a la parte determinista de la misma, que vamos a denotar $d(p)$, y otra considerada como un término del error, que vamos a denotar como U. El término del error es la parte estocástica de la función de demanda y puede tener distintas distribuciones de probabilidad.

Comunmente, la funcion de demanda se expresa en estas dos partes de dos formas:

- Modelo aditivo: $X(p) = d(p) + U$.
- Modelo multiplicativo: $X(p) = d(p)\,U$

La función de demanda $X(p)$ está definida en el intervalo $[p_{min}, p_{max}]$, siendo p_{min} el mínimo precio que tiene sentido poner al producto (podría ser el coste del mismo) y p_{max} el mayor precio con el que alguien podría comprar el producto.

En cuanto a la función del error U, hay que tener en cuenta para su análisis que la demanda no puede ser negativa, para ello la probabilidad de que el error sea mayor que la parte determinista de la demanda sea cero. Nosotros en la práctica no vamos a considerar la propia distribución del error U, sino que vamos a usar $U^* =: \max(U, -d(p))$.

Una parte fundamental de la demanda es la elasticidad. La elasticidad de la demanda corresponde con la variación en la demanda producida por un pequeño cambio en el precio, es decir

$$\varepsilon(p) := \frac{\partial d(p)}{\partial p}$$

A la hora de analizar el beneficio que va a generar la venta de un producto es importante tener en cuenta el número de unidades que se tiene. De esta forma si consideramos que se tiene y unidades de ese producto disponibles para la venta, aunque la demanda sea más de y no las podremos vender, por

lo tanto la demanda no es el único factor a tener en cuenta para maximizar el beneficio. La siguiente fórmula corresponde con las ventas esperadas para un nivel de stock y precio dado:

$$S(p,y) := \int_0^y [1-F(p,x)]\,dx.$$

Con este valor de las ventas esperadas podemos definir el beneficio esperado.

$$B(p,y) := pS(p,y)$$

Del mismo modo que para analizar la demanda hemos definido su elasticidad respecto al precio, se pueden definir dos nuevas elasticidades sobre las ventas esperadas.

- La elasticidad de las ventas esperadas respecto del precio:

$$\varepsilon^p(p,y) := \frac{-pS_p(p,y)}{S(p,y)} \quad \text{donde } S_p := \frac{\partial S}{\partial p}. \tag{1.1}$$

  Conforme aumenta el precio disminuyen las ventas esperadas, así que $\varepsilon^p(p,y)$ mide ese descenso.

- La elasticidad de las ventas esperadas respecto del nivel de stock:

$$\varepsilon^y(p,y) := \frac{yS_y(p,y)}{S(p,y)} \quad \text{donde } S_y := \frac{\partial S}{\partial y}.$$

  Conforme aumenta el nivel de stock aumentan las ventas esperadas, así que $\varepsilon^y(p,y)$ es una medida de ese crecimiento.

El objetivo es encontrar el precio que maximice el beneficio, así que el precio óptimo, p^*, será aquel que maximize $S(p,y)$ para un determinado $y > 0$. Si la función de distribución $F(p,x)$ de la variable aleatoria de la demanda es convexa entonces esta garantizada la unicidad del precio óptimo. Sin embargo, se puede garantizar la unicidad de la solución con menos restricciones sobre la función de demanda.

Proposición 1.1. *Si $\varepsilon^p(p,y)$ aumenta al aumentar p, entonces:*

- *La solución a $\frac{\partial}{\partial p}B(p,y) = 0$ es el único precio óptimo y además $p^*(y)$ satisface $\varepsilon^p(p^*(y),y) = 1$.*

Demostración. Tenemos que encontrar el máximo en p de la función $B(p,y)$, por (1.1):

$$\frac{\partial}{\partial p}B(p,y) = S(p,y) + pS_p(p,y) = (1-\varepsilon^p(p,y))\int_0^y [1-F(p,x)]\,dx = 0.$$

Ahora bien, como $\int_0^y [1-F(p,x)]\,dx$ siempre es positivo, la única forma de que se anule la ecuación es que $\varepsilon^p(p,y) = 1$, y como es creciente a lo sumo lo hará en un punto, p^*. □

Si añadimos un valor salvable el precio optimo aumentará, sin embargo si asumimos cierta penalización por no vender el precio óptimo disminuirá.

Proposición 1.2. *El precio óptimo disminuye si le añadimos un coste de almacenamiento.*

Demostración. Si consideramos un coste de almacenamiento $h > 0$ para las unidades no vendidas, la función de beneficio tiene la siguiente forma:

$$B(p,y) = p\int_0^y [1-F(p,x)]\,dx - h\int_y^\infty [1-F(p,x)]\,dx.$$

Derivando respecto del precio y gracias a (1.1) tenemos:

$$\frac{\partial}{\partial p}B(p,y) = S(p,y)[1-\varepsilon^p(p,y)] - \int_y^\infty h[1-F_p(p,x)]\,dx.$$

Si tomamos $p = p^*$ (precio óptimo sin tener en cuenta el coste de almacenamiento), por la proposición anterior sabemos que la primera parte de la derivada es cero. Como la segunda parte es una integral positiva, tenemos que $\frac{\partial}{\partial p}B(p^*,y) < 0$ y por lo tanto el beneficio esta decreciendo en ese punto, así que el precio óptimo en este caso tendrá que ser menor que p^*. □

1.3. Modelos analizados

En [12] se puede ver un estudio de los modelos de demanda más comunes. En el presente trabajo vamos a trabajar con tres de los modelos de demanda más comunes, estos modelos pueden ser tomados de forma determinista o con un término aleatorio.

- **Modelo lineal.** Es el modelo más simple y su versión determinista tiene la siguiente forma:

$$d(p) = a - bp \text{ con } a, b > 0. \tag{1.2}$$

Veamos la forma de la función de demanda.

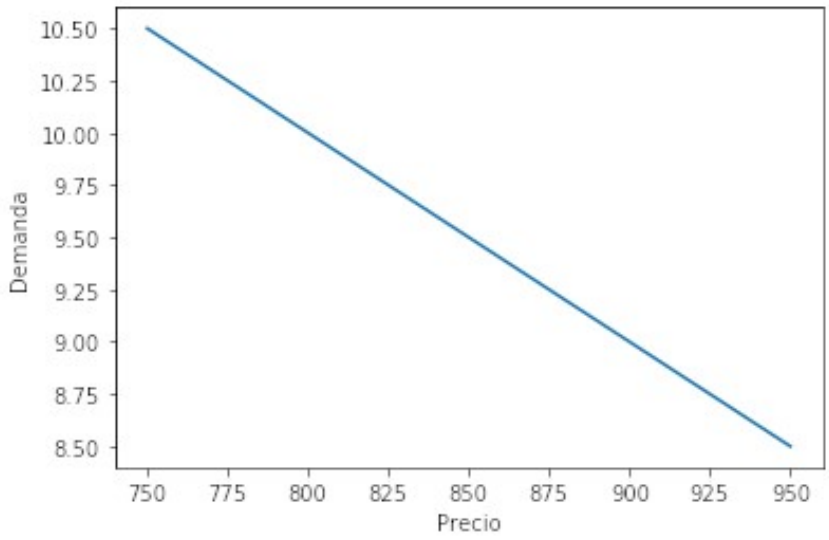

Figura 1.1: Demanda lineal con $a = 18, b = 0{,}01$.

La función de beneficio si se tiene un stock suficiente para satisfacer a la demanda es $B(p) = ap - bp^2$ por lo tanto el precio óptimo sería $p^* = -a/b$, solución a la ecuación $B'(p) = 0$.

Con esta demanda suele ser utilizado un modelo aditivo del término aleatorio.

- **Modelo Isoelástico.** Es un modelo potencial, su versión determinista tiene la siguiente forma:

$$d(p) = ap^{-b} \text{ con } a > 0 \text{ y } b > 1. \tag{1.3}$$

Veamos la forma de la función de demanda en el rango de precios $[340, 420]$.

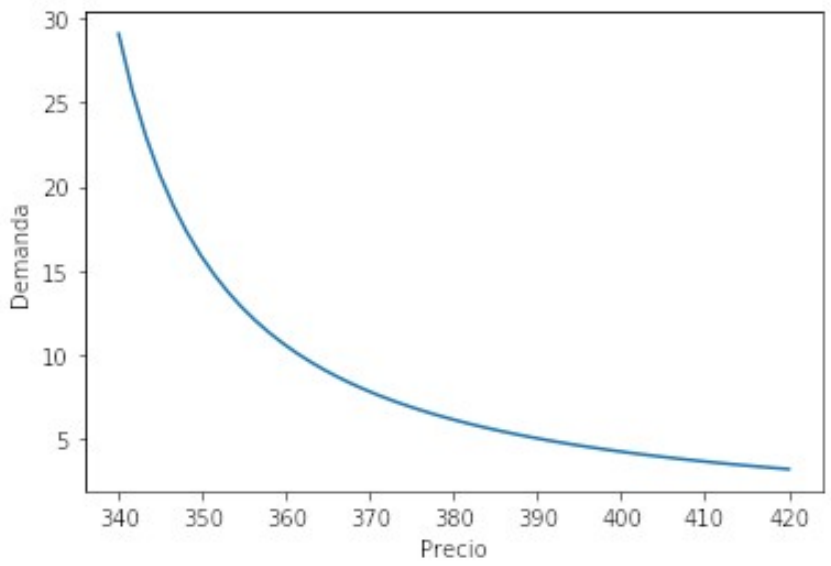

Figura 1.2: Demanda isoelástica con $a = 750, b = 1{,}2$

Con esta demanda suele ser utilizado un modelo multiplicativo del término aleatorio.

- **Modelo Exponencial.** Su versión determinista tiene la siguiente forma:

$$d(p) = a\exp(-bp) \text{ con } a, b > 0. \tag{1.4}$$

Veamos la forma de la función de demanda.

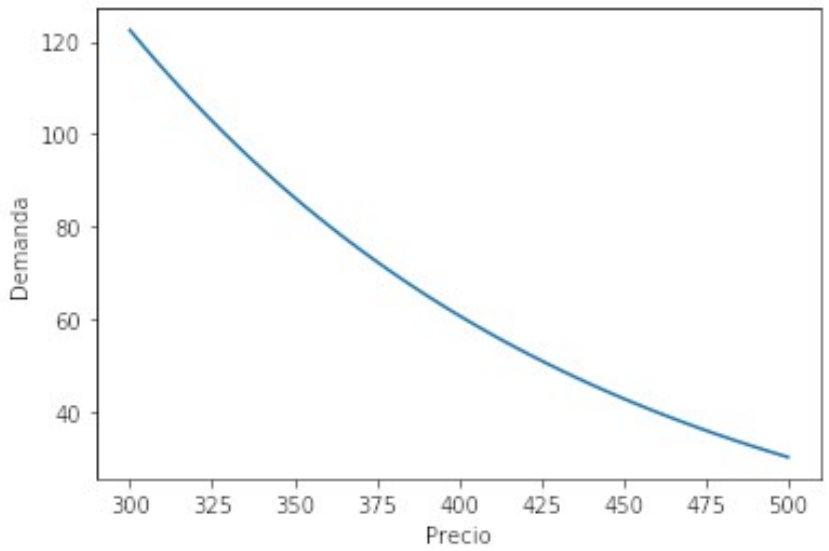

Figura 1.3: Demanda exponencial con $a = 1000, b = 0{,}007$

La función de beneficio si se tiene un stock suficiente para satisfacer a la demanda es $B(p) = ap\exp(-bp)$ por lo tanto el precio óptimo sería $p^* = 1/b$, solución a la ecuación $B'(p) = 0$. Con esta demanda suele ser utilizado un modelo multiplicativo del término aleatorio.

Además, pese a que no los vamos a utilizar en el capítulo 4, vamos a ver la forma de dos modelos de demanda también bastante comunes.

- **Modelo logarítmico.** Su versión determinista tiene la siguiente forma:

$$d(p) = \gamma \log(a - bp) \text{ con } \gamma, a, b > 0.$$

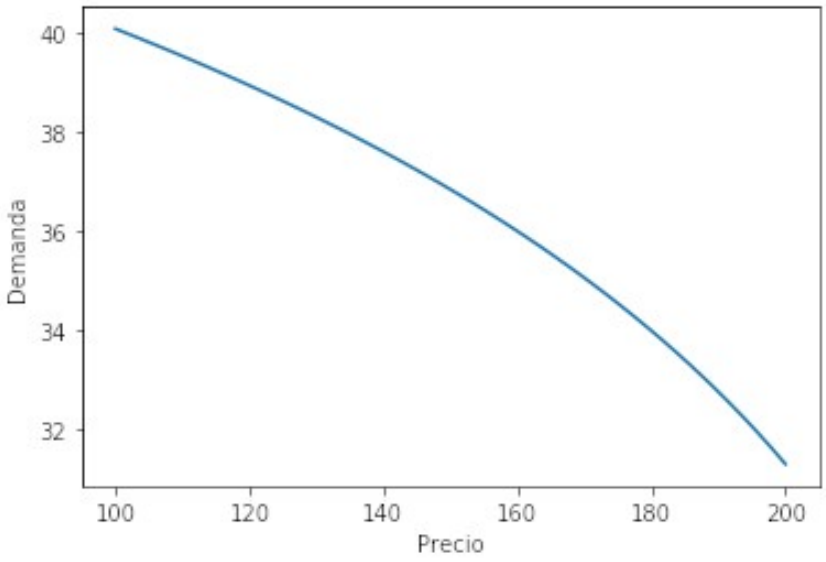

Figura 1.4: Demanda logarítmica con $\gamma = 8$, $a = 250$ y $b = 1$.

- **Modelo logit.** Su versión determinista tiene la siguiente forma:

$$d(p) = \frac{a\exp(-bp)}{1+\exp(-bp)} \text{ con } a, b > 0.$$

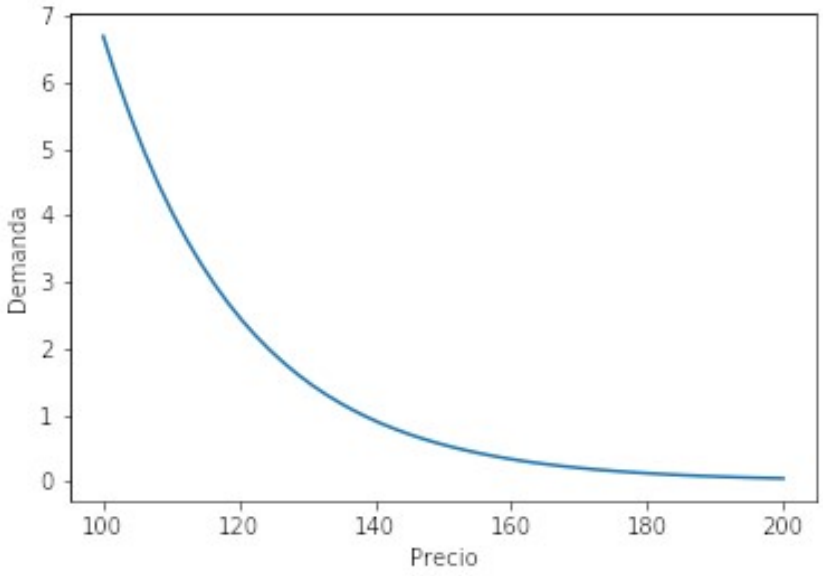

Figura 1.5: Demanda logit con $a = 1000$ y $b = 0{,}05$.

Capítulo 2

Aprendizaje por refuerzo

2.1. Introduccion

El aprendizaje por refuerzo es una de las 3 principales areas del aprendizaje automático (machine learning), junto al aprendizaje supervisado y al aprendizaje no supervisado. Mientras que en el aprendizaje supervisado un agente aprende con el conocimiento proporcionado por el exterior y en el no supervisado el agente trata de encontrar patrones en la información que ya conoce, en estos métodos el agente aprende por prueba y error interactuando con el entorno y obteniendo así información. El aprendizaje por refuerzo estudia como un agente debe tomar una serie de decisiones en un entorno con el fin de lograr un objetivo. Cabe destacar que el término aprendizaje por refuerzo puede referirse a este campo del aprendizaje automático, a los problemas de este campo y a los métodos que tratan de resolverlos. En [27] se puede ver una amplia introducción al campo del aprendizaje por refuerzo así como numerosos ejemplos relacionados.

En el aprendizaje por refuerzo, una agente interactúa con el medio, tomando una serie de acciones (decisiones) y recibiendo la recompensa asociada a ellas. El objetivo para este agente consiste en maximizar de alguna manera las recompensas obtenidas a lo largo del proceso. El proceso puede estar limitado de alguna forma, tanto en un numero de pasos determinado, un estado final tal que la llegada a este supone el final del proceso, etc.

El aprendizaje por refuerzo tiene una potente aplicación a juegos, de forma que se ha hecho muy popular en la consecución de estrategias óptimas para juegos de una casuistica muy compleja como el Go, llegando incluso a generar estrategias que derrotan a los mejores jugadores del mundo. Veamos brevemente como se podría adaptar el aprendizaje por refuerzo a un juego más sencillo como una partida de ajedrez. El agente es el jugador que, en base a la posición de las figuras en el tablero, interactua con el entorno a través de sus movimientos (decisiones) y que trata de aprender en base a recompensas, ganar o perder finalmente la partida. A través de las experiencias previas, el agente va mejorando sus decisiones y va aprendiendo a "jugar" al ajedrez. En el artículo [23] se muestra un algoritmo de este tipo.

2.2. Problema de decisión de Markov

Vamos a describir el problema de aprendizaje por refuerzo apoyándonos en la estructura de los procesos de decisión de Markov, estos procesos nos permiten describir matemáticamente el problema, de manera que se puede desarrollar una teoría y unos métodos para abordarlo y solucionarlo. En [15] se abordan los procesos de decisión de Markov desde un punto de vista riguroso y más matemático.

El pretexto con el que empezamos a trabajar, dado que estamos en un problema de aprendizaje por refuerzo, consiste en un agente que interactúa con su entorno con el fin de lograr un objetivo. Todo lo que rodea al agente es considerado el entorno, este puede variar por cuestiones ajenas al agente o por las decisiones que este toma a lo largo del tiempo.

A partir de aquí se pueden definir una serie de conceptos necesarios para este tipo de procesos y problemas. La repetición completa del proceso define un **episodio** (época) y este es sólo una parte del

proceso de aprendizaje por refuerzo, ya que en la práctica son necesarias muchas repeticiones para poder llegar a aprender del problema. Cada episodio se puede llevar a cabo en unos pasos determinados, refiriéndonos al número de veces que el agente interactúa con el entorno. Por tanto podemos definir el conjunto de pasos como el conjunto discreto de valores de tiempo: $T := \{1,2,..,t,t+1..\}$ en los cuales se produce las interacciones y también los cambios del entorno. Así que tenemos que asumir que el entorno permanece invariante durante cada paso y que el conjunto de pasos es discreto. Tomadas estas consideraciones, veamos los elementos principales de un proceso de decisión de Markov.

- **Estado**. El estado consiste en la información del entorno que percibe el agente en cada paso. Denotamos al espacio de estados como $S := \{s^1, s^2, ..., s^n\}$. Se va a trabajar con procesos de Markov finitos, es decir, que tienen un número de estados finitos, ya que es la única posibilidad de llevarlo a cabo en la práctica. A lo largo del trabajo, cuando no sea importante conocer el estado en cuestión, se denotará s_t como el estado alcanzado en el paso t, por ejemplo s_t podría referirse a que en el paso t el agente está en el estado s^1.

- **Acción.** Corresponde con la decisión tomada por el agente en un paso determinado. Definimos el conjunto de acciones que se pueden llevar a cabo durante todo el proceso como $A := \{a^1, a^2, .., a^m\}$. Si consideramos que estamos en el estado s^i, el espacio de acciones asociado a ese estado se denota como $A(s^i)$, este espacio representa las posibles decisiones del agente cuando está en el estado s^i. Tenemos en cuenta la misma consideración para a_t que con s_t, cuando hablemos de los acciones de forma general, a_t será la acción tomada en el paso t, $t \in T$.

- **Recompensa.** Es un valor, que a menudo suele ser estocástico, que recibe el agente en cada paso del proceso en función del estado en el que se encuentra y la acción que toma. El objetivo del agente en el proceso de aprendizaje por refuerzo es el de maximizar la recompensa recibida a lo largo del mismo, por lo tanto las recompensas son básicas para la resolución de este tipo de problemas. Denotamos $\mathscr{R}$ como el espacio de posibles recompensas durante el proceso y r_t como la recompensa obtenida en el paso t para todo $t \in T$.

- **Política.** La política es la base del aprendizaje del agente y corresponde con una función que define su forma de actuar. Esta función hace corresponder a cada estado en el que se encuentra el agente con una serie de probabilidades asociadas a las acciones a tomar por parte del mismo. Denotamos como π la política que sigue un agente a lo largo del proceso, esta política viene determinada por $\pi(s,a)$ con $s \in S$ y $s \in A(s)$, que corresponde con la probabilidad de que el agente tome la acción a si está en el estado s. Añadir que si la política es determinista, $\pi(s)$ nos indica la acción que la política toma en el estado $s \in S$, es decir

$$\pi(s) = \max_{a \in A(s)} \pi(s,a) \text{ para todo } s \in S.$$

El proceso básico que tiene lugar en cada paso $t \in T$ se podría resumir de la siguiente forma: el agente esta en el estado $s_t \in S$ y siguiendo una política π elige una acción $a_t \in A(s_t)$. Esta acción le lleva a recibir una recompensa r_t y a transitar al estado s_{t+1}, donde el agente tomará otra decisión y se repetirá de forma iterativa el mismo proceso.

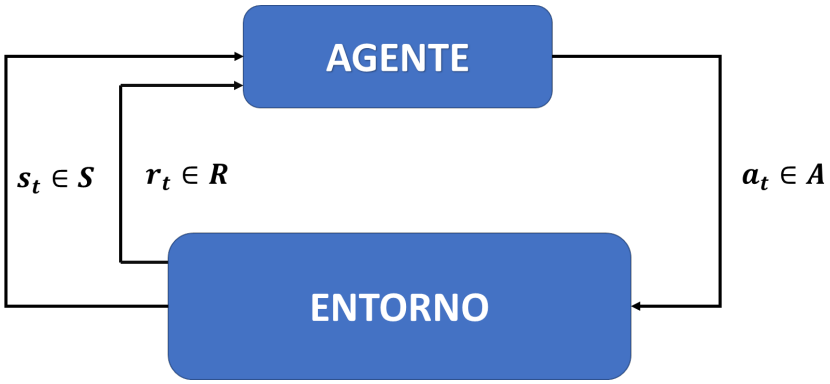

Figura 2.1: Paso en tiempo t.

Hay que destacar que cada problema de aprendizaje por refuerzo es distinto y aunque a primera vista esta estructura pueda parecer simple, es muy flexible y se adapta bien a modelar situaciones reales. Por ejemplo, no hemos comentado cuando un episodio termina, se puede fijar un número de pasos para cada episodio o se puede establecer un estado terminal, de tal manera que al llegar el agente a ese estado se pone fin al episodio. También se puede aplicar un criterio de parada del proceso dependiente de las recompensas obtenidas a lo largo del mismo, etc.

Para definir formalmente el proceso de decisión de Markov, asuminos que en cierta manera existe un modelo por el que se rige el entorno. De esta forma existen unas probabilidades de transición de un estado a otro en función de la información del entorno (tanto la actual como la del pasado). Es decir, existe

$$P(s_{t+1} = s' | s_t, a_t, s_{t-1}, ..., a_1) \text{ para todo } s_t \in S, a_t \in A, t \in T.$$

Esto representa entonces la probabilidad de que, habiéndose visitado los estados correspondientes y habiendo tomado determinadas acciones en los t primeros pasos, el siguiente estado en ser visitado sea $s_{t+1} = s'$.

Los procesos de decisión de Markov se basan en la propiedad de Markov, esta nos indica que esta probabilidad sólo depende del paso anterior, es decir

$$P(s_{t+1} = s' | s_t, a_t, s_{t-1}, ..., a_1) = P(s_{t+1} = s' | s_t, a_t) \text{ para todo } s_t \in S, a_t \in A, t \in T.$$

Así que para que el proceso que estamos describiendo sea un proceso de decisión de Markov se debe cumplir esta propiedad. Por tanto asumimos que la propiedad de Markov se cumple en nuestros problemas de aprendizaje por refuerzo. Formará parte de la modelización del problema adecuarlo a un Problema de decisión de Markov. Un ejemplo de esta adaptación podría ser incluir información de los estados anteriores en el estado actual, de forma que la probabilidad de transición sólo dependería del estado y acción actual, sin embargo el estado actual tendría información de estados pasados.

Definición 1. Teniendo en cuenta las consideraciones anteriores podemos definir un proceso de decisión de Markov como la terna (S, A, P), donde P es la función de probabilidades de transición definida de la siguiente forma:

$$\begin{aligned} P : S \times A \times S &\longrightarrow [0,1] \\ (s, a, s') &\longmapsto P(s'|s,a) \end{aligned}$$

Ahora que ya tenemos definido un modelo para nuestro entorno, es necesario cuantificar la bondad de una política en función de los beneficios que se esperan obtener siguiéndola a lo largo del episodio. Como nuestro objetivo es maximizar el valor esperado de las recompensas totales, aparece el concepto de recompensa esperada a partir de un determinado estado $s \in S$ o paso $t \in T$. La forma más intuitiva de esta recompensa esperada es la siguiente:

$$R(s_t) = E\left[\sum_{k=0}^{\infty} r_{t+k} | s_t\right] \text{ para todo } t \in T.$$

Observar que esta recompensa depende directamente del estado actual, por lo que si queremos ver la recompensa esperada de todo el proceso habría que tomar el estado inicial, s_1. Con idea de generalizar esta definición y para poder darle más importancia a las recompensas más próximas se introduce el término **factor de descuento**. El factor de descuento, $\gamma \in [0,1]$ determina el valor actual de futuras recompensas. El valor de una recompensa k pasos después tendrá un valor actual de γ^{k-1} veces su valor natural (el que tendría si fuera percibida en el paso actual). Observar que si $\gamma = 1$ estamos en el caso de la definición anterior.

$$R(s_t) = E^{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t\right] \text{ para todo } t \in T. \tag{2.1}$$

Aunque es lo más común tratar de maximizar la recompensa esperada, también se pueden poner otras metas y por tanto otra función objetivo. Un caso de esto sería el modelo de recompensa media, que trata de maximizar la media de las recompensas a largo plazo. En este caso la función objetivo sería:

$$R(s_t) = \lim_{n\to\infty} E\left[\frac{1}{n}\sum_{k=0}^{n} r_{t+k} | s_t\right] \text{ para todo } t \in T.$$

En nuestro caso vamos a usar la definición que tiene en cuenta el factor de descuento (2.1) ya que es la más extendida en el campo del aprendizaje por refuerzo.

Una vez hemos visto cual es el objetivo de los problemas de aprendizaje por refuerzo, vamos a estudiar las funciones de valor, que no son más que el beneficio esperado siguiendo una política determinada. Estas funciones dependen de políticas, estados y/o acciones pues tienen en cuenta valores futuros de las recompensas.

Definición 2. La función de valor-estado, $V^{\pi}(s)$, es una función que muestra la recompensa esperada cuando se parte del estado s y se sigue la política π.

Al final, este valor no es más que una predicción de las recompensas esperadas a lo largo del proceso. Como se trata de una estimación, es más costoso y por lo tanto requiere de más episodios para poder asignar una función de valor a cada estado. Sin embargo, este concepto es más usado en los métodos de resolución que las recompensas propiamente (r_t) ya que no sólo tiene en cuenta la recompensa del paso correspondiente sino que tienen en cuenta las recompensas que recibirá el agente a largo plazo. Suponiendo que E^{π} denota a la esperanza de una determinada variable aleatoria si se sigue la política π en cada paso, podemos definir matemáticamente la función de **valor de estado** de la siguiente forma:

$$V^{\pi}(s_t) := E^{\pi}\left[R(s_t)\right] = E^{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t\right]$$

Del mismo modo podemos definir otra función que también sirve a la hora de optimizar políticas, es la llamada función de **valor-acción**, también conocida como la función Q.

Definición 3. La función de valor-acción, $Q^{\pi}(s,a)$, corresponde con la recompensa esperada comenzando en el estado s, tomando la acción a y siguiendo la política π en los siguientes pasos.

Matemáticamente se puede expresar como

$$Q^{\pi}(s_t,a_t) := E^{\pi}\left[R(s_t)|a_t\right] = E^{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t, a_t\right]$$

Como hemos comentado estos valores se pueden estimar a través de la experiencia del agente, y a la hora de poder calcularlos, es importante una propiedad de recursividad que cumplen, la llamada

ecuación de Bellmann.

$$\begin{aligned}
V^{\pi}(s) &= E^{\pi}\left[\sum_{k=0}^{\infty}\gamma^{k} r_{t+k} | s_t = s\right] \\
&= E^{\pi}\left[r_t + \gamma\sum_{k=0}^{\infty}\gamma^{k} r_{t+k+1} | s_t = s\right] \\
&= \sum_{a\in A(s)}\pi(s,a)\sum_{s'\in S}P(s'|s,a)\left[R(s,a,s') + \gamma E^{\pi}\left[\sum_{k=0}^{\infty}\gamma^{k} r_{t+k+1} | s_{t+1} = s'\right]\right] \\
&= \sum_{a\in A(s)}\pi(s,a)\sum_{s'\in S}P(s'|s,a)\left[R(s,a,s') + \gamma V^{\pi}(s')\right]
\end{aligned}$$

Denotando $R(s,a,s') := E[r_t | s_t = s, a_t = a, s_{t+1} = s']$ para algún $t \in T$. Por tanto la ecuación de Bellman para la función de valor $V^{\pi}(s)$ tiene la siguiente forma:

$$V^{\pi}(s) = \sum_{a\in A(s)}\pi(s,a)\sum_{s'\in S}P(s'|s,a)\left[R(s,a,s') + \gamma V^{\pi}(s')\right]. \tag{2.2}$$

Aplicando el mismo razonamiento a la función de valor-acción Q, y teniendo en cuenta que en este caso la acción está elegida ya para el paso actual, se llega a la siguiente ecuación:

$$Q^{\pi}(s,a) = \sum_{s'\in S}P(s'|s,a)\left[R(s,a,s') + \gamma V^{\pi}(s')\right] \tag{2.3}$$

Por tanto, teniendo en cuenta las ecuaciones de Bellman (2.2) y (2.3), llegamos a una relación directa entre ambas funciones de valor.

$$V^{\pi}(s) = \sum_{a\in A(s)}\pi(s,a)\,Q^{\pi}(s,a) \tag{2.4}$$

Merece la pena comentar que para la ecuación de optimalidad de Bellman existe siempre una única solución, que en nuestro caso corresponderá con el valor de estado correspondiente, ya sea V o Q. Por tanto esta propiedad nos permite trabajar con estas relaciones de recursividad a la hora de resolver los problemas de aprendizaje por refuerzo. Para resolverlos hay que hallar la política óptima en base a estos valores así que hay que estudiar los valores máximos de estas funciones. Comenzamos estableciendo un criterio de comparación entre políticas, denotaremos como $\pi \geq \pi'$ si π es más óptima o al menos igual que la política π'.

$$\pi \geq \pi' \Leftrightarrow V^{\pi}(s) \geq V^{\pi'}(s) \text{ para todo } s \in S.$$

Por construcción siempre habrá una política que cumple esto, la política óptima, que denotamos como π^*. Podemos definir sus funciones de valor asociadas:

$$V^{\pi^*}(s) = \max_{\pi} V^{\pi}(s), \forall s \in S.$$

$$Q^{\pi^*}(s,a) = \max_{\pi} Q^{\pi}(s,a), \forall s \in S, \forall a \in A.$$

Ahora bien, a partir de (2.4) se observa que se debe cumplir:

$$V^{\pi^*}(s) = \sum_{a\in A(s)}\pi^*(s,a)\,Q^{\pi^*}(s,a) \leq \max_{a\in A(s)} Q^{\pi^*}(s,a) \text{ para todo } s \in S.$$

Pero es que V^{π^*} debe ser mayor o igual que $\max_{a\in A(s)} Q^{\pi^*}(s,a)$ pues es el máximo y si no la acción a haría posible una política con un valor de la función de estado mayor. Así que se tiene la siguiente igualdad:

$$V^{*}(s) = \max_{a\in A(s)} Q(s,a) \text{ para todo } s \in S.$$

Teniendo en consideración esta última igualdad (2.2), y la ecuación de recursividad de Bellman para la función Q, (2.3), si evaluamos en la política óptima π^* se tienen estas dos ecuaciones:

$$V^*(s) = \max_{a \in A(s)} P(s'|s=s, a=a)\left[R(s,a,s') + \gamma V^*(s')\right] \tag{2.5}$$

$$Q^*(s,a) = \sum_{s' \in S} P(s'|s=s, a=a)\left[R(s,a,s') + \gamma \max_a Q^*(s',a')\right]$$

Estas ecuaciones se conocen como las **ecuaciones de optimalidad de Bellman** y nos proporcional una solución analítica a la hora de encontrar la política óptima de nuestro problema.

2.3. Métodos de resolución.

Una vez hemos establecido una estructura matemática con la que trabajar en los problemas de aprendizaje por refuerzo, estamos en disposición de resolverlos. Vamos a estudiar diversos métodos y técnicas para la resolución de los procesos de decisión de Markov y por tanto de nuestro problema.

Hemos visto que el conocimiento de las funciones de valor óptimas nos proporciona una política óptima, sin embargo, normalmente estas funciones de valor no son conocidas y hay que estimarlas para encontrar las políticas óptimas a partir de ellas. Los algoritmos suelen tener dos fases: una en la que se estiman las funciones de valor para los diferentes estados $V(s)$ o estados/acciones $Q(s,a)$, y otra en la que a partir de esos valores se encuentra la política óptima π.

Generalmente se diferencian dos tipos de métodos para dar con la política de decisión óptima, la que supone mayor recompensa esperada.

- **Métodos basados en el modelo.** Son métodos donde el conocimiento del modelo es usado para encontrar la política óptima, es decir, se tienen en cuenta los parámetros del modelo para la obtención de las funciones de valor. En nuestro caso, como estamos ante un problema de decisión de Markov, estos parámetros se refieren a las probabilidades de transición entre los estados $P(s'|s,a)$ y a la función de recompensas $R(s,a,s')$. Con el conocimiento de estas dos funciones se puede describir el funcionamiento del entorno en el que se encuentra el agente.

  Por tanto, para este tipo de métodos es fundamental la estimación de los parámetros del modelo (o el conocimiento previo de ellos). Esta estimación se puede llevar a cabo mediante la experiencia que obtiene el agente al interactuar con su entorno.

  Una vez estimados estos parámetros, se acostumbra a usar programación dinámica en este tipo de métodos. Esta técnicas llegan a la política óptima a través de las ecuaciones de optimalidad de Bellmann (2.5). Los dos principales algoritmos de programación dinámica aplicada a este contexto son el de iteración de políticas y el de iteración de valores.

  En el algoritmo de iteración de valores, se obtiene la política óptima después de estimar la función de valor $V(s)$, mientras que en el de iteración de políticas se obtiene la política óptima directamente a través de las políticas π.

- **Métodos sin uso del modelo.** Son métodos en los que no se tiene el cuenta el modelo para la resolución del problema, es decir, se pueden llevar a cabo sin conocen las probabilidades de transición entre estados ni la función de recompensa.

  Por tanto, este tipo de métodos trata de estimar las funciones de valor directamente gracias a la experiencia del agente, recompensas obtenidas a lo largo de los pasos, y a partir de ahí encontrar la política óptima.

En este trabajo vamos a analizar los principales algoritmos que no hacen uso del modelo del entorno, estos son los más habituales en el campo del aprendizaje por refuerzo ya que se suele asumir que el modelo es desconocido.

2.3.1. Métodos de diferencia temporal.

En el campo del aprendizaje por refuerzo, y dentro de los métodos que no hacen uso del modelo, los algoritmos más utilizados son los que estiman las funciones de valor siguiendo el método de diferencia temporal. Estos algoritmos son los que se van a tratar en el trabajo. No obstante, antes de explicar estas técnicas vamos a comentar un poco los métodos Monte-Carlo.

Los métodos de Monte-Carlo, tal y como los de diferencia temporal, usan la experiencia para abordar el problema de estimación. Los métodos Monte-Carlo esperan hasta ver la recompensa al final del episodio para actualizar las funciones de valor, esta estimación de las funciones de valor a lo largo del proceso iterativo puede tener la siguiente forma:

$$V^{\pi}(s_t) = V^{\pi}(s_t) + \alpha\left[R_t - V^{\pi}(s_t)\right]$$

Donde R_t es la recompensa que se obtiene al final del episodio siguiendo una determinada política π (contando desde el paso t) y α es una constante.

Vamos a centrarnos ahora en los métodos de diferencia temporal, en este caso también hacen uso de la experiencia para la estimación, sin embargo, sólo tienen que esperar hasta el paso siguiente para estimar la función de valor correspondiente. Es decir, usan las estimaciones de las funciones de valor de otros estados para la estimación del actual, esto se conoce como *bootstraping* y es una característica que comparten con los algoritmos de programación dinámica.

La forma más simple de los algoritmos de diferencia temporal es conocida como $TD(0)$ y actualiza las estimaciones de la siguiente manera:

$$V^{\pi}(s_t) = V^{\pi}(s_t) + \alpha\left[r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)\right]$$

El método primero estima la función de valor como $r_{t+1} + \gamma V^{\pi}(s_{t+1})$ y luego actualiza la estimación anterior en base a este valor y al factor de aprendizaje α, α es por tanto una constante que indica el grado de credibilidad de la estimación respecto al valor actual. Esta es la base de actualización de los métodos de diferencia temporal.

La principal diferencia con los métodos derivados de la programación dinámica es que estos usan las estimaciones de todos los posibles futuros estados (*full backup*), mientras que la diferencia temporal sólo usa las del siguiente estado (*sample backup*).

Si se quiere estimar las funciones de valor estado asociadas a la política π, (V^{π}), la estructura del algoritmo es la siguiente:

Algorithm 1 Diferencia Temporal, $T(0)$

1: Inicializar $V(s)$ para todo $s \in S$
2: **loop** (Para cada episodio)
3: Inicializar s_1
4: **loop** (Para cada paso $t = 1, 2, ..,$)
5: Elegir a_t con la función de probabilidad $\pi(s_t, a)$
6: Tomar acción a_t, obtener r_t y s_{t+1}
7: $V^{\pi}(s_t) = V^{\pi}(s_t) + \alpha\left[r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)\right]$
8: $s_t = s_{t+1}$

Notar que es posible que haya que implementar un criterio de parada según la estructura del problema.

Está probado en [26] que para que se asegure la convergencia de un algoritmo de diferencia temporal se tiene que visitar cada estado infinitas veces y el factor de aprendizaje α tiene que ir decreciendo a lo largo del tiempo.

Tenemos entonces que en estos métodos, para poder garantizar su convergencia hay que recorrer los estados (o los pares estado-acción) infinitas veces, lo cual no es posible. Esto nos lleva a plantearnos políticas eficientes con las que recorrer los estados de forma que nos podamos acercar a los valores

óptimos. Parece claro que las los estados (o pares estado-acción) que más interesa estimar bien son los que tienen el mayor valor, sin embargo hay que tener cuidado a la hora de recorrerlos para no quedarse estancados en políticas subóptimas.

Por tanto, el dilema es si ir adquiriendo información para lograr más precisión en las estimaciones de las funciones de valor mayores (explotación) o tratar de estudiar nuevos estados (exploración). En la práctica se usa una política que combine ambos conceptos. Vamos a ver dos de las principales políticas que aúnan estos dos conceptos:

- **Políticas ε greedy.** Son las políticas de comportamiento más usadas en el campo del aprendizaje por refuerzo. Diferencian la acción que maximiza las funciones de valor para el estado dado, de forma que considerando $a' = argmax_{a\in\mathscr{A}}(Q(s,a))$ y $0 \leq \varepsilon \leq 1$:

$$\pi(s,a) = \begin{cases} \varepsilon/|A| & \text{si } a \neq a' \\ 1-\varepsilon+\varepsilon/|A| & \text{si } a = a' \end{cases}$$

- **Políticas Softmax.** Este tipo de políticas no dan la misma probabilidad al resto de acciones que no son a', aunque siguen dando la mayor probabilidad a a'. Si consideramos $\delta > 0$ el parámetro de exploración, la política de este tipo más usada es de la siguiente forma:

$$\pi(s,a) = \frac{\exp(Q(s,a)/\delta)}{\sum_{a'\in A}\exp(Q(s,a')/\delta)} \text{ para todo } s \in S, a \in A.$$

Hemos visto que esta forma nos permite estimar las funciones de valor. Siguiendo este método podemos definir dos principales tipos de algoritmos para la resolución de nuestro problema. Cabe destacar que, pese a que hemos visto el caso general con la función de valor de estado, los algoritmos de este tipo que vamos a ver están implementados con la función de valor acción, Q, ya que es mucho más sencillo ver que acción proporciona mayor función de valor en cada paso.

- **Algoritmos *on-policy*.** En ellos la política evaluada es la misma que se usa para recorrer y estimar las funciones de valor.
- **Algoritmos *off-policy*.** La política a evaluar no tiene porque ser la misma que la que se usa para recorrer los estados ya que no es necesario una política para recorrer los estados.

Estos dos conceptos se verán mejor con los 4 algorítmos que vamos a estudiar a continuación, dos *on-policy* y otros dos *off-policy*. Los dos primeros que vamos a ver se pueden consultar en [19], mientras que los dos últimos se ven en [27].

2.3.2. Algoritmo Q-Learning

El algoritmo del Q learning es un algoritmo off-policy y es probablemente el más popular dentro del campo del aprendizaje por refuerzo debido a su sencilla implementación, fue introducido por Watkins en 1989 [30].

Hace uso de las funciones de valor-acción $Q(s,a)$, que va estimando a la vez que determina la política óptima asociada a ellas. La actualización de las funciones de valor tiene la forma de la diferencia temporal:

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha_t\left[r_t + \gamma \max_{a\in A} Q(s_{t+1},a) - Q(s_t,a_t)\right]$$

Vemos que $r_t + \gamma \text{máx}_{a\in A} Q(s_{t+1},a)$ actúa como la estimación de $Q(s_t,a_t)$. Además vemos que el factor de aprendizaje α puede tener un valor distinto para cada paso.

En cuanto a la convergencia del método se puede ver en [31] que :

Proposición 2.1. *$Q(s,a) \to Q^{\pi^*}(s,a)$ para todo $s \in S, a \in A$ si cada par de estado-acción es visitado infinitas veces y además $\sum_{i=0}^{\infty}\alpha_i = \infty$ y $\sum_{i=0}^{\infty}\alpha_i^2 < \infty$.*

Por lo tanto conviene tomar los α_t decrecientes a lo largo del episodio. La estructura del algoritmo es la siguiente:

Algorithm 2 Q-Learning

Inicializar $Q(s,a)$ para todo $s \in S$, $a \in A$
loop (Para cada episodio)
    Inicializar s_1
    loop (Para cada paso $t = 1,2,..,)$
        Elegir a_t con una política derivada de Q.
        Tomar acción a_t, obtener r_t y s_{t+1}
        $Q(s_t,a_t) = Q(s_t,a_t) + \alpha_t \left[r_t + \gamma \max_{a\in A} Q(s_{t+1},a) - Q(s_t,a_t)\right]$
        $s_t = s_{t+1}$
    end loop
end loop
$\pi(s) = \arg\max_{a\in A} Q(s,a)$ para todo $s \in S$

2.3.3. Algoritmo SARSA

El algoritmo SARSA es un algoritmo on-policy, es decir necesita de una politica para recorrer los estados y así actualizar las funciones de valor-acción. El nombre deriva de las siglas que resumen el proceso que se lleva a cabo en cada paso de este algoritmo, *State, Action, Reward, State, Action*. La regla de actualización es la siguiente:

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha_t \left[r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right]$$

Observar que en este caso la estimación que se hace de la función de valor es $r_t + \gamma Q(s_{t+1},a_{t+1})$, por lo tanto es necesario hacer uso de una política para determinar cual es la acción a_{t+1} elegida.

Respecto a la convergencia, se puede ver en [24] que el algoritmo converje si cada par estado-acción es visitado infinitas veces y la política utilizada tiende a la política que determina a_t como $a_t = \arg\max_{a\in A}(Q(s_t,a))$ para todo $s \in S$, $a_t \in A$.

Si tomamos una política derivada de la función de valor acción, la estructura del algoritmo es la siguiente:

Algorithm 3 SARSA

Inicializar $Q(s,a)$ para todo $s \in S$, $a \in A$
loop (Para cada episodio)
    Inicializar s_1
    Elegir a_1 con una política derivada de Q.
    loop (Para cada paso $t = 1,2,..,)$
        Tomar acción a_t, obtener r_t y s_{t+1}
        Elegir a_{t+1} con una política derivada de Q.
        $Q(s_t,a_t) = Q(s_t,a_t) + \alpha_t \left[r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right]$
        $s_t = s_{t+1}$ y $a_t = a_{t+1}$
    end loop
end loop
$\pi(s) = \arg\max_{a\in A} Q(s,a)$ para todo $s \in S$

2.3.4. Algoritmo de Doble Q-Learning

Si analizamos la forma de estimación de las funciones de valor de los métodos anteriores podemos observar que el uso del máximo supone cierto optimismo a la hora de estimar las funciones de valor, esto se traduce en un sesgo para la estimación. Para intentar tratar con esto, el algoritmo doble Q-learning

combina dos Q valores para cada par de estado acción de forma que se reduce este sesgo positivo. Como deriva del Q-Learning, es un algoritmo *off-policy*.

En cada visita a un estado-acción se elije la función de Q valor a actualizar de forma aleatoria, así que la estructura del algoritmo es:

Algorithm 4 Doble Q-Learning

Inicializar $Q_1(s,a)$ y $Q_2(s,a)$ para todo $s \in S, a \in A$
loop (Para cada episodio)
    Inicializar s_1
    loop (Para cada paso $t = 1,2,..,)$
        Elegir a_t con una política derivada de $Q_1 + Q_2$.
        Tomar acción a_t, obtener r_t y s_{t+1}
        if numero aleatorio $\leq 0{,}5$ **then**
            $Q_1(s_t,a_t) = Q_1(s_t,a_t) + \alpha_t \left[r_t + \gamma Q_2(s_{t+1}, \arg\max_{a\in A} Q_1(s_{t+1},a)) - Q_1(s_t,a_t)\right]$
        else
            $Q_2(s_t,a_t) = Q_2(s_t,a_t) + \alpha_t \left[r_t + \gamma Q_1(s_{t+1}, \arg\max_{a\in A} Q_2(s_{t+1},a)) - Q_2(s_t,a_t)\right]$
        end if
        $s_t = s_{t+1}$
    end loop
end loop
$\pi(s) = \arg\max_{a\in A} Q(s,a)$ para todo $s \in S$

2.3.5. Algoritmo *Expected* SARSA

Siguiendo la misma idea que el algoritmo de doble Q-learning, el expected SARSA trata de evitar el optimismo generado por la elección de la acción que proporciona el máximo Q valor. En este caso este algoritmo modifica al SARSA y tiene en cuenta el retorno esperado en lugar del máximo retorno esperado de la siguiente manera:

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha_t \left[r_t + \gamma \sum_{a\in A} \pi(s_{t+1},a)\, Q(s_{t+1},a) - Q(s_t,a_t)\right] \tag{2.6}$$

Teniendo en cuenta esta regla de actualización, la estructura que sigue el algoritmo es igual que la de SARSA, por tanto es un algoritmo *on-policy*. Usando este algoritmo se puede reducir el sesgo pero también supone un mayor coste computacional respecto al SARSA.

Capítulo 3

Aprendizaje por refuerzo multi-agente.

En el capítulo anterior hemos visto como se aplica el aprendizaje por refuerzo a problemas donde hay un único agente, sin embargo, también se puede adecuar a problemas donde hay varios agentes tomando acciones en un entorno. La principal diferencia con los métodos de un sólo agente es que las decisiones del resto de los agentes influyen en el entorno y por tanto hay que estudiar de alguna manera el comportamientos de los otros agentes para optimizar sus intereses.

Esta estructura del problema nos lleva directamente a la teoría de juegos, un área muy estudiada de la economía donde se analiza el comportamiento de varios individuos que compiten o cooperan tomando una serie de decisiones en un entorno llamado juego. Se puede modelar el juego de manera que se mantienen las definiciones y notaciones de los procesos de decisión de Markov, sin embargo, las acciones y las recompensas no tienen porque ser las mismas para cada agente. Así el proceso general de aprendizaje es similar, se parte de un determinado estado s_t, cada agente i toma una decisión a_t^i, recibe una recompensa r_t^i y el juego cambia al estado s_{t+1} donde se repite el proceso. Añadir además que se considera el espacio de estados finito y que los agentes toman las decisiones en cada paso de forma simultánea.

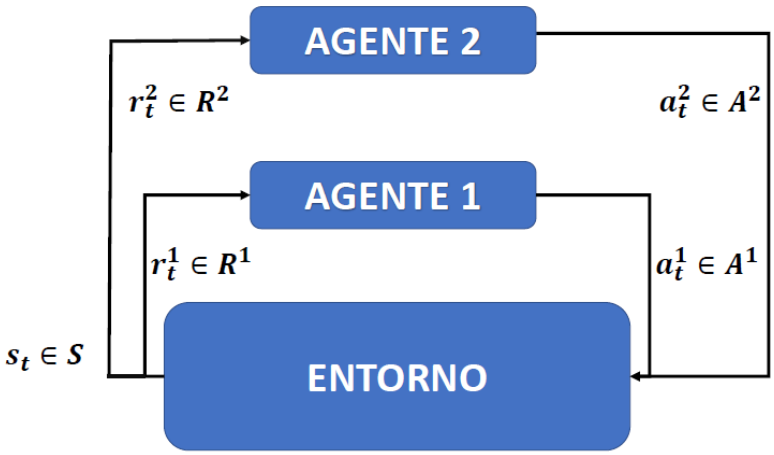

Figura 3.1: Paso en tiempo t (aprendizaje con 2 agentes).

Del mismo modo que el aprendizaje por refuerzo con un sólo agente se basa en el contexto de los Problemas de decisión de Markov, la estructura de los problemas multi-agente se basa en los llamados **juegos estocásticos**, que no son más que una generalización de los Procesos de Decisión de Markov. En el caso más simple vamos a estudiar los juegos estocásticos con un único estado, los llamados juegos matriciales, y luego introduciremos la forma general de los juegos estocásticos, tal y como se expone en [11] y [19].

La aplicación de estos métodos es muy amplia ya que permiten modelar muchas situaciones reales donde pueden llegar a intervenir muchos agentes. Por ejemplo, en [1] se puede ver una aplicación del

aprendizaje por refuerzo multi-agente a un problema de tráfico en una ciudad, donde cada vehículo interacciona en el juego (entorno) como un agente. En [5], en cambio se puede ver la competencia entre dos empresas de comercio electrónico en función de sus decisiones de fijación de precios. Muchos de los ejemplos reales introducen redes neuronales para la estimación de las funciones de valor ya que el tratar con varios agentes aumenta mucho el coste computacional de los métodos. Es por eso que en el capítulo se van a ver métodos más simples desde el punto de vista computacional que podremos aplicar de manera eficaz a un bajo número de agentes.

3.1. Juegos Matriciales.

Los juegos matriciales se desarrollan en un entorno que tiene un único estado y donde hay varios agentes tomando decisiones sobre él. Como sólo hay un estado, los juegos matriciales vienen definidos por la tupla $(n, A^1, .., A^n, R^1, ..R^n)$ donde n es el número de agentes, A^i el espacio de posibles acciones a tomar por el agente i y R^i la función de recompensa asociada al agente i:

$$R^i : A^1 \times A^2 \times .. \times A^n \longrightarrow \mathbb{R} \qquad 0 \leq i \leq n.$$

Observar que se mantienen los conceptos del aprendizaje con un único agente, sin embargo, al haber varios agentes interactuando, las funciones de recompensa no sólo dependen de las acciones del propio agente sino también de las del resto. Además podemos definir el espacio conjunto de acciones como $A := A^1 \times A^2 \times .. \times A^n$.

El caso particular de dos agentes, $n = 2$, nos lleva a los llamados juego de dos oponentes. En estos juegos dos matrices de recompensas definen el juego, las filas corresponden con las posibles acciones a tomar por parte del Agente 1 y las columnas representan las acciones que puede tomar el Agente 2. Por tanto el juego matricial viene definido por $M^1, M^2 \in \mathscr{M}(|A^1|, |A^2|)$.

Existen multitud de juegos de dos oponentes como el clásico juego de piedra papel o tijera o el dilema del prisionero. Otro ejemplo de este tipo de juegos podría ser el llamado juego de halcón-paloma (puede consultarse en [20]), donde dos agentes se enfrentan por la posesión de un objeto de valor $v > 0$. Se pueden comportar de manera agresiva (halcón) o pacífica (paloma), si ambos se comportan de manera agresiva se enzarzan en una pelea que les supone unos costes c, si ambos se comportan de manera pacífica se reparten el objeto y si uno se comporta pacíficamente y el otro de forma agresiva, se queda el objeto el agente agresivo. Este juego queda definido por la siguiente tabla:

		Agente 2	
		Pacífico	Agresivo
Agente 1	Pacífico	$V/2$, $V/2$	0, V
	Agresivo	V, 0	$V/2-C$, $V/2-C$

Cuadro 3.1: Juego de Halcón-Paloma.

De la misma forma que en los problemas de único agente era fundamental el concepto de política, aquí aparece el concepto de estrategia, que determina la forma en la que determinado agente se comporta en el juego, es decir, determina la acción que toma el agente.

Definición 4. Llamamos **estrategia** a una función de probabilidad σ_i que asocia a cada acción de A^i una probabilidad de ser tomada, $\sigma_i : A^i \longrightarrow [0,1]$ con $1 \leq i \leq n$.

El conjunto de las estrategias de todos los agentes conforma la estrategia conjunta del juego, σ. Normalmente es necesario ver la estrategia conjunta desde el punto de vista de un agente particular, por ello denotamos σ_{-i} al conjunto de estrategias del resto de los agentes que no son el agente i. Siguiendo esta notación, la estrategia conjunta del juego es $\sigma = \langle \sigma_i, \sigma_{-i} \rangle$. Se pueden definir dos tipos distintos de estrategias:

- Las estrategias puras, son las que son deterministas, es decir, la función σ_i relaciona todas acciones menos una a una probabilidad cero. Por lo tanto la estrategia viene unívocamente definida por una acción.
- Las estrategias mixtas, que son las estrategias estocásticas, es decir, la función σ_i no es trivial.

En la teoría de juegos el concepto de equilibrio de Nash es esencial de forma que puede ser considerado una solución al juego.

Definición 5. Llamamos equilibrio de Nash puro al conjunto de acciones $(a_*^1,..,a_*^n)$ que cumplen:

$$\begin{aligned} R^1(a_*^1,a_*^2,..,a_*^n) &\geq R^1(a^1,a_*^2,..,a_*^n) \text{ para todo } a^1 \in A^1 \\ &\vdots \\ R^1(a_*^1,a_*^2,..,a_*^n) &\geq R^1(a_*^1,a_*^2,..,a^n) \text{ para todo } a^n \in A^n. \end{aligned}$$

Notar que si tenemos estrategias mixtas esta definición de equilibrio de Nash no nos vale, para llegar a la definición general, primero definimos matemáticamente la recompensa esperada tomando una determinada estrategia conjunta.

$$R^i(\sigma) = \sum_{a=a^1\times..\times a^n\in A} R^i(a)\prod_{j=1}^{n}\sigma_j(a^j)$$

A la hora de definir el **equilibrio de Nash**, aparece el concepto de estrategia de mejor respuesta.

Definición 6. Llamamos **estrategia de mejor respuesta** a la estrategia σ_i que, dada una estrategia σ_{-i}, genera mayor recompensa esperada. Es decir σ_i^* es estrategia de mejor respuesta a σ_{-i}^* si:

$$R^i(\langle\sigma_i^*,\sigma_{-i}^*\rangle) \geq R^i(\langle\sigma_i,\sigma_{-i}^*\rangle) \quad \text{para toda posible estrategia } \sigma_i.$$

Definición 7. Llamamos equilibrio de Nash al conjunto de estrategias donde cada una de ellas es una estrategia de mejor respuesta sobre el conjunto del resto de estrategias, es decir:

$$\begin{aligned} R^1(\langle\sigma_1^*,\sigma_{-1}^*\rangle) &\geq R^1(\langle\sigma_1,\sigma_{-1}^*\rangle) \quad \text{para toda posible estrategia } \sigma_1. \\ &\vdots \\ R^n(\langle\sigma_n^*,\sigma_{-n}^*\rangle) &\geq R^n(\langle\sigma_n,\sigma_{-n}^*\rangle) \quad \text{para toda posible estrategia } \sigma_n. \end{aligned}$$

Observando esta definición de equilibrio de Nash vemos que siguiendo uno de ellos, ninguno de los agentes puede mejorar su recompensa cambiando su estrategia. El estudio de equilibrios de Nash con estrategías mixtas está justificado en que existen resultados para demostrar la existencia de estos, sin embargo, no existen resultados clásicos para el caso de equilibrios de Nash puros. En [18] se puede ver una demostración del siguiente resultado:

Teorema 3.1 (Teorema de Nash)**.** *En todo juego matricial existe al menos un equilibrio de Nash.*

Veamos ahora los distintos tipos de juegos matriciales, usualmente se clasifican en:

- Juegos de suma cero. Son juegos de dos oponentes donde $M_1 = -M_2$, es decir, las recompensas son opuestas para los dos agentes. Son equivalentes a los llamados juegos de suma constante, donde para cada par de acciones la suma de las recompensas de los dos agentes es constante.

  Un ejemplo de este tipo es el juego de piedra, papel o tijera. En el juego sólo hay tres resultados posibles, victoria derrota o empate y viene definido por la siguiente matriz de recompensas:

		Agente 2		
		Piedra	Papel	Tijeras
Agente 1	Piedra	$(0,0)$	$(-1,1)$	$(1,-1)$
	Papel	$(1,-1)$	$(0,0)$	$(-1,1)$
	Tijeras	$(-1,1)$	$(1,-1)$	$(0,0)$

Cuadro 3.2: Juego de piedra, papel o tijera.

En este tipo de juegos los equilibrios de Nash tienen la forma de los llamados equilibrios adversarios, que cumplen:

$$R^1(\langle \sigma_1^*, \sigma_2^* \rangle) \leq R^1(\langle \sigma_1^*, \sigma_2 \rangle) \quad \text{para toda posible estrategia } \sigma_2.$$
$$R^2(\langle \sigma_1^*, \sigma_2^* \rangle) \leq R^2(\langle \sigma_1^*, \sigma_2 \rangle) \quad \text{para toda posible estrategia } \sigma_1.$$

Esto significa que tomando la estrategia del equilibrio adversario, un agente nunca empeora su recompensa esperada por mucho que el otro agente cambie la estrategia. Así que podríamos catalogar este equilibrio como un óptimo en el peor de los casos.

- Juegos de equipo. En este tipo de juegos matriciales la recompensas son identicas para cada agente, por lo tanto el objetivo es maximizar la recompensa de cualquier agente y se puede reducir a un problema con un único agente que toma varias acciones, es decir, su espacio de acciones es $A = A^1 \times .. \times A^n$. En este tipo de juegos aparece el llamado equilibrio de coordinación, que no es más que el equilibrio de Nash que supone mayor recompensa esperada.

- Juegos generales. Así son considerados el resto de juegos matriciales, Aunque esta garantizada la existencia de algún equilibrio, puede no ser fácil dar con él. En el caso de juegos de dos oponentes, existe un método de programación cuadrática para llegar al equilibrio tal y como se detalla en [17].

3.2. Juegos estocásticos

Los juegos estocásticos son una generalización de los juegos matriciales y de los Procesos de decisión de Markov. Generalizan a los juegos matriciales en el aspecto de tener varios estados en el entorno, en concreto el espacio de estados se denota como S, y en cada uno de esos estados la estructura es la de un juego matricial, por lo tanto un juego estocástico es una sucesión de juegos matriciales conectados por unas determinadas probabilidades de transición. Además estos juegos también generalizan a los Procesos de decisión de Markov ya que tenemos n agentes decidiendo en lugar de uno sólo.

Siguiendo las notaciones de la sección anterior y del capítulo anterior, tenemos que un juego estocástico viene definido por la tupla $(n, S, A, R^1, ..R^n, P)$, donde P es la función de probabilidades de transición entre estados y está definida de la siguiente forma:

$$P : S \times A \times S \longrightarrow [0,1]$$
$$(s, a^1, .., a^n, s') \longmapsto P(s'|s, a^1, .., a^n)$$

Además de la misma manera que aparece la política en los procesos de decisión de Markov, es esencial en los juegos estocásticos.

Definición 8. En el contexto de los juegos estocásticos llamamos política a una colección de distribuciones de probabilidad que determinan la forma de actuar de un agente en cada estado. Por tanto una política es una colección de estrategias, una para cada estado.

Llamamos política conjunta del juego a una colección políticas, una correspondiente a cada agente. Si queremos denotar una política conjunta haciendo énfasis en un agente determinado, se escribe:

$$\pi = \langle \pi_i, \pi_{-i} \rangle$$

Donde π_{-i} denota a colección de las políticas de todos los agentes menos el i.

El objetivo de cada agente es similar al del los procesos de decisión de Markov, maximizar su recompensa esperada, por tanto hay que tener en cuenta un modelo para esta recompensa esperada. En nuestro caso y tal y como hemos hecho en el capítulo anterior vamos a usar la recompensa con factor

de descuento (2.1) ya que es el modelo más común también en el caso de varios agentes. Teniendo en cuenta este modelo, las funciones de valor se pueden definir de la siguiente forma:

$$V_i^{\pi}(s_t) := E^{\pi}\left[R^i(s_t)\right] = E^{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k}^i | s_t\right].$$

Donde V_i^{π} representa la función de valor de estado del agente i siguiendo la política conjunta π.

$$Q_i^{\pi}(s_t, a_t) := E^{\pi}\left[R(s_t)|a_t\right] = E^{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k}^i | s_t, a_t\right].$$

En este caso Q_i^{π} corresponde con la función de valor-acción del agente i para la política conjunta π. Al igual que en los juegos matriciales, la bondad de una política particular no se puede evaluar sin tener en cuenta las políticas del resto de los agentes, es por ello que hay que hacer uso de las llamadas políticas de mejor respuesta y de los equilibrios de Nash.

Definición 9. En el contexto de juegos estocásticos llamamos política de mejor respuesta repecto a la colección de políticas del resto de agentes π_{-i}^* a la política π_i^* que cumple:

$$V_i^{\langle \pi_i^*, \pi_{-i}^* \rangle}(s) \geq V_i^{\langle \pi_i, \pi_{-i}^* \rangle}(s) \text{ para todo } s \in S \text{ y toda posible política } \pi_i.$$

Definición 10. En el contexto de juegos estocásticos decimos que la colección de políticas π^* es un equilibrio de Nash si cumple:

$$\begin{aligned} V_1^{\pi^*}(s) &\geq V_1^{\langle \pi_1, \pi_{-1}^* \rangle}(s) \text{ para todo } s \in S \text{ y toda posible política } \pi_1. \\ &\vdots \\ V_n^{\pi^*}(s) &\geq V_n^{\langle \pi_n, \pi_{-n}^* \rangle}(s) \text{ para todo } s \in S \text{ y toda posible política } \pi_n. \end{aligned}$$

El alto número de agentes aumenta la complejidad del problema sustancialmente, es por eso que no hay muchos resultados para un número alto de agentes. Si consideramos el caso de dos agentes en un juego estocástico, se puede ver en [8] el siguiente resultado de existencia de equilibrios de Nash.

Teorema 3.2. *En todo juego estocástico de dos agentes existe al menos un equilibrio de Nash.*

Los distintos tipos de juegos estocásticos son similares a los tipos de los juegos matriciales:

- Juegos de suma cero. Están formados por juegos matriciales de suma cero en cada uno de los estados.
- Juegos de equipo. Están formados por juegos matriciales de equipo en todos sus estados.
- Juegos repetidos. Son un caso degenerado de juegos estocásticos para un sólo estado, se juegan una y otra vez por un tiempo determinado.
- Juegos generales. El resto de juegos estocásticos no incluidos en ningunas de estas categorías.

3.3. Algorítmos multiagente

En este tipo de problemas, el concepto de solución es relativo, pues si queremos maximizar la recompensa esperada de un agente del juego, esta depende directamente de las politicas que siguen el resto. Así que antes de estudiar los métodos resolutivos hay que concretar en cierta manera el concepto de solución. Por un lado podemos buscar la política óptima para una política conjunta π_{-i}, e intentar maximizar la recompensa esperada respecto a esa política, estos son los llamados Métodos de mejor respuesta. Y por otro lado podemos buscar los equilibrios de Nash del problema de manera que nos garanticemos la mayor recompensa esperada independientemente de las políticas que sigan el resto de agentes.

3.3.1. Métodos de mejor respuesta.

Estos métodos intentan llegar a una política óptima π_i^* en base a una política conjunta dada π_{-i}. Para que estos métodos se puedan llevar a cabo esta política conjunta debe ser una política estacionaria (o que converja a una política estacionaria), es decir, que no varie a lo largo del proceso de aprendizaje. La ventaja de estos métodos es que puede que la política π_{-i} no sea una política de mejor respuesta y por lo tanto el método puede suponer mayor recompensa esperada que los métodos de equilibrios. Como estos métodos necesitan de políticas estacionarias en cierta manera es fundamental estimarlas o conocerlas totalmente. Veamos distintos métodos:

- **Métodos de aprendizaje con un único agente.** Cuando el resto de los agentes siguen políticas estacionarias, el juego estocastico se puede tranformar en un proceso de decision de Markov. En particular el espacio de estados se mantiene y las acciones que toman el resto de agentes (a través de sus políticas estacionarias) determinan las probabilidades de transición entre estados y la función de recompensa del agente pricipal. Además el espacio de acciones pasará de ser n-dimensional a ser unidimensional.

  En estas circustancias se pueden aplicar algorítmos de aprendizaje con un agente como el Q-Learning o el SARSA.

- **Métodos de acción conjunta.** Son métodos que se aplican a juegos de equipo, donde todos los agentes reciben las mismas recompensas. Como se tienen las mismas recompensas estos juegos son similares a los procesos de decisión de Markov con un sólo agente, salvo que el espacio de acciones es n-dimensional. El problema reside en que el resto de los agentes no tienen porque estar coordinados ni tomar las mismas acciones. Por lo tanto es necesario estimar la política que siguen.

- **Opponent modelling.** Se trata de un algoritmo donde los agentes se dividen en dos grupos, el agente principal y los oponentes. Se basa en la estimación de las políticas de los oponentes a través de la experiencia para poder obtener las funciones de valor del principal. La estimación de las políticas de los oponentes se hace de la siguiente forma:

$$\pi_{-i}(a^{-i}) = \frac{n(s,a^{-i})}{n(s)} \text{ para todo } a^{-i} \in A^1 \times .. \times A^{i-1} \times A^{i+1} \times .. \times A^n.$$

  Si consideramos $n(s)$ el número de veces que se ha pasado por el estado $s \in S$ a lo largo del proceso y $n(s,a^{-i})$ el número de veces que estando en s se han tomado las acciones conjuntas a^{-i}.

Algorithm 5 Opponent modelling

Inicializar $Q(s,\langle a^i,a^{-i}\rangle)$, $n(s)$ y $n(s,a^{-i})$ para todo $s \in S$, $\langle a^i,a^{-i}\rangle \in A$
loop (Para cada episodio)
  Inicializar s_1
  loop (Para cada paso $t = 1,2,..,$)
    $O(s,a_t^i) = \sum_{a^{-i}\in A^{-i}} \frac{n(s,a^{-i})}{n(s)} Q(s,\langle a_t^i,a^{-i}\rangle)$
    Elegir a_t con una política derivada de O.
    Tomar acción a_t^i, obtener r_t, s_{t+1} y a_t^{-i}
    $Q(s,\langle a_t^i,a_t^{-i}\rangle) = Q(s,\langle a_t^i,a_t^{-i}\rangle) + \alpha_t\left[r_t^i + \gamma \max_{a\in A^i} O(s_{t+1},a) - Q(s,\langle a_t^i,a_t^{-i}\rangle)\right]$
    $s_t = s_{t+1}$
    $n(s,a_t^{-i}) = n(s,a_t^{-i}) + 1$
    $n(s) = n(s) + 1$
  end loop
end loop
$\pi(s) = \arg\max_{a\in A^i} O(s,a)$ para todo $s \in S$

3.3.2. Métodos de equilibrios.

Estos métodos se basan en los la obtención de los equilibrios de Nash del juego, de manera que se pueda maximizar la recompensa esperada sea cual sea la política del resto. Para obtener el equilibrio de Nash del juego completo se basan en la obtención del equilibrio de Nash para cada estado $s \in S$, es decir el equilibrio de Nash de cada juego matricial, y así obtener la función de valor estado.

Algorithm 6 Método de equilibrios

Inicializar $Q(s,a)$ para todo $s \in S$, $a \in A$
loop (Para cada episodio)
  Inicializar s_1
  loop (Para cada paso $t = 1,2,..,)$
    Elegir a_t con una política derivada de Q.
    Tomar acción a_t, obtener r_t, s_{t+1} y a_t^{-i}
    $V(s_{t+1}) =$ función de valor-estado siguiendo la política del equilibrio de Nash
    loop (Para cada agente $i = 1,2,..,n)$
      $Q_i(s,\langle a_t^i, a_t^{-i}\rangle) = Q_i(s,\langle a_t^i, a_t^{-i}\rangle) + \alpha_t\left[r_{t+1}^i + \gamma V_i(s_{t+1}) - Q_i(s,\langle a_t^i, a_t^{-i}\rangle)\right]$
    end loop
    $s_t = s_{t+1}$
  end loop
end loop

La ventaja que presentan respecto a los métodos de mejor respuesta es que no suponen políticas estacionarias del resto, que es lo más común en problemas reales ya que no es muy acertado pensar que el resto de los agentes no aprende a su manera a lo largo del proceso.

En general como es compleja la obtención de los equilibrios, estos métodos se usan en problemas relativamente pequeños o de un determinado tipo como pueden ser los juegos de suma cero, juegos de dos personas, etc. Veamos dos algoritmos de este tipo:

- **Minimax-Q**. Se usa con juegos de suma cero ya que en estos está garantizada la presencia de un sólo equilibrio de Nash. La obtención de funciones de valor se basa primero en minimizar la recompensa del oponente para luego maximizar la propia, ya que las recompensas son opuestas.

$$V(s) = \max_{\pi} \min_{a^{-i} \in A^{-i}} \sum_{a^i \in A^i} \pi(s,a^i) Q(s,\langle a^i, a^{-i}\rangle)$$

- **Nash-Q.** Se ocupa de juegos estocásticos generales cumpliendo unas determinadas condiciones bastante estrictas. Todos los juegos matriciales del juego estocástico deben tener un único equilibrio de Nash y los equilibrios además deben cumplir una serie de restricciones.

  Bajo estas condiciones el método establece las funciones de valor a través de un método de programación cuadrática.

Capítulo 4

Aplicación a Tarificación Dinámica. Implementación y Resultados.

En este capítulo vamos a ver la aplicación práctica de lo visto en los anteriores capítulos, para ello veremos primero una aplicación del aprendizaje por refuerzo de un único agente al *dynamic pricing* con los algoritmos (2), (3), (4) y (2.6). Posteriormente veremos un pequeño ejemplo para dos agentes en el *dynamic pricing* con la implementación del algoritmo *Opponent Modelling* (5).

Destacar que estos algoritmos se han desarrollado con el lenguaje de programación python [21] sin el uso de ninguna librería especifica de aprendizaje por refuerzo, simplemente creando clases según los distintos entornos de cada problema y programando los algoritmos sobre objetos de esas clases. Se han utilizado las siguientes librerías generales de python 3: *numpy*, *matplotlib*, *pandas* y *random*.

4.1. Aprendizaje con único agente.

El problema que vamos a estudiar viene planteado gracias a la empresa de Instrumentación y Componentes Inycom S.A. en uno de sus proyectos reales con una empresa externa. Si bien es verdad que la ausencia de datos de ventas nos ha hecho tener que estimar la demanda con funciones generales, los precios y ventas aproximadas si que coinciden con las de los productos reales.

El problema está basado en un negocio de venta de piezas de aeronaves de segunda mano que vende varias piezas (productos) a lo largo de un año. El proceso de adquisición de las piezas es el siguiente, el negocio adquiere una aeronave muy usada y es desmontada por piezas para poder reutilizarlas en el negocio después de una certificación. Una vez superadas estas certificaciones, las piezas son puestas en el mercado de segunda mano.

A la hora de modelar este problema real, vamos a tener en cuenta una serie de consideraciones, vamos a suponer que disponemos de un cierto número de piezas después de desmantelar un avión. Después de las certificaciones sucesivas, estas piezas son puestas en el mercado durante un año (lo que están en vigor las certificaciones), por lo tanto toda pieza no vendida en esos 12 meses no supondrá ningún beneficio para el negocio. El objetivo es fijar precios de venta para cada uno de esos meses con el fin de maximizar el beneficio total. Añadir que existe un coste de almacenamiento de las piezas $h > 0$ por lo que es más beneficioso para el negocio vender una pieza en el mes 1 que en el mes 12. Vamos a trabajar con tres productos o piezas distintos.

- **Producto A.** Corresponde con un cableado de la aeronave, se disponen de 105 unidades al principio del proceso y tiene un coste de almacenamiento de $h = 5$ u.m. por mes y unidad. El rango de precios que pueden ser establecidos es $p \in [725, 975]$ y además la demanda de este producto es lineal.

$$d(p) = 18 - 0{,}01p.$$

  Si consideramos la demanda estocástica, esta es aditiva con un término del error distribuido como

una normal $N(0,3/2)$, de forma que:

$$d(p) = 18 - 0{,}01p + N(0,3/2).$$

- **Producto B.** Son sensores de temperatura de la aeronave, se disponen de 800 unidades al principio del proceso y tiene un coste de almacenamiento de $h = 2$ u.m. por mes y unidad. El rango de precios en el que está definida la demanda y por tanto podemos establecer es $p \in [350,450]$. La demanda de este producto es exponencial.

  $$d(p) = 1000\exp(-0{,}007p).$$

  Además si consideramos la demanda estocástica, esta es multiplicativa y el término del error sigue una normal $N(1,1/8)$, de forma que:

  $$d(p) = 1000\exp(-0{,}007p) * N(1,1/8).$$

- **Producto C.** Corresponde con los paneles test de la aeronave, se disponen de 250 unidades al principio y tienen un coste de almacenamiento de $h = 5$ u.m. por mes y unidad. El rango de precios es $p \in [340,420]$ y la demanda de este producto es isoelástica.

  $$d(p) = 750(p - 325)^{-1,2}.$$

  Además la demanda estocástica es multiplicativa con un término del error que sigue una distribución exponencial $Exp(1)$, de forma que:

  $$d(p) = 750(p - 325)^{-1,2} * Exp(1).$$

En esta situación vamos a modelar el problema como un proceso de decisión de Markov y le vamos a aplicar los algoritmos estudiados. El estado viene definido por el nivel de stock (número de piezas disponibles) y el periodo en el que se está (mes 1, mes 2, etc). Respecto a las probabilidades de transición, el nivel de stock es simplemente el del periodo anterior menos lo que se ha vendido en el anterior mes (en función de la demanda). Las acciones del problema son los distintos precios a fijar en cada mes y las recompensas en cada mes corresponden con el beneficio de las ventas de ese mes. Destacar que las funciones de demanda son iguales en todos los meses, no se ha añadido ninguna componente estacional dada la naturaleza del problema. Esto es debido a que los meses no siempre son fijos y depende de la época del año en la que se obtenga la aeronave para ser vendida por piezas.

Los algoritmos nos proporcionan una estrategia de fijación de precios para todos los estados (tomando la política óptima determinista) y además nos proporcionan la información de la recompensa esperada al tomar cierta decisión en cierto estado gracias a la tabla de funciones de valor-acción Q. Por lo tanto no sólo nos dan un único camino de óptimo de precios sino que nos proporcionan más información lo que nos supone una mayor flexibilidad para tomar otras decisiones. Destacar que la política óptima la fijaremos en base al beneficio anual.

En todos los algorítmos se usa un ε para las políticas greedy de 0,8 que decrece hasta 0 conforme avanzan las épocas, la elección de este valor de ε está justificada ya que nos permite explorar suficientes políticas a lo largo del proceso. Respecto al factor de aprendizaje α, no hemos fijado ninguno en concreto sino que en cada algoritmo se ha elegido el que mejor se adaptaba, destacar que este valor depende directamente del número de épocas. Veremos una comparación de resultados obtenidos variando el factor de aprendizaje sólo para un caso. El factor de descuento fijado es $\gamma = 1$ ya que nos centramos en el beneficio general del año. Vamos a entrenar los modelos primero según la demanda determinista y los evaluaremos con la demanda estocástica asociada. Luego los entrenaremos directamente con las demandas estocásticas y analizaremos las diferencias observadas.

4.1.1. Producto A.

Primero vemos la elección del factor de aprendizaje del modelo, estas comparaciones no las volveremos a hacer para cada producto y algoritmo. Destacar que el espacio de acciones discreto va desde $p = 725$ u.m. hasta $p = 975$ u.m. de 25 en 25, por tanto tenemos 11 posibles acciones a tomar. Llevamos a cabo el Q-Learning con el fin de maximizar el beneficio anual obtenido al vender el producto A, para ello realizamos 20000 épocas/episodios (se ha visto que más no tienen por qué ser necesarias), los resultados obtenidos según el factor de aprendizaje son los siguientes:

Factor de aprendizaje	Recompensa esperada	Factor de aprendizaje	Recompensa esperada
$\alpha = 1$	94.980	$\alpha = 0,5$	92.730
$\alpha = 0,9$	94.980	$\alpha = 0,4$	92.725
$\alpha = 0,8$	94.670	$\alpha = 0,3$	93.020
$\alpha = 0,7$	94.235	$\alpha = 0,2$	92.705
$\alpha = 0,6$	93.487	$\alpha = 0,1$	89.880

Cuadro 4.1: Resultados obtenidos tras Q-Learning (en u.m.).

Por tanto vemos razonable elegir un factor de aprendizaje de 1 en este caso. Se podría elegir otro factor de aprendizaje y otro número de épocas y es probable que se obtuvieran los mismos resultados, pero este nos garantiza llegar al valor óptimo en un menor número de épocas que el resto.

Ajustamos los cuatro algoritmos para maximizar el beneficio anual obtenido al vender el producto A. Entrenando los algoritmos con la función de demanda determinista obtenemos los siguientes resultados:

	Recompensa obtenida	Meses empleados	Épocas
Q-Learning	94.980	12	20.000
SARSA	93.350	12	80.000
Doble Q-Learning	93.625	12	25.000
Expected SARSA	92.955	12	35.000

Cuadro 4.2: Resultados obtenidos para el producto A.

Los cuatro algoritmos obtienen beneficios bastante similares quizás es debido a la relativa simplicidad del problema determinista, sin embargo, el Q-Learning es el que más destaca y parece que ha dado un óptimo que puede ser global. En los cuatro algoritmos se vende el producto durante los 12 meses, es decir, ninguna de las políticas de fijación de precios provoca que el stock de productos se acabe antes. Respecto a las épocas vemos que el que más ha empleado es el SARSA, pero no son realmente importantes en ente caso puesto que se ha intentado maximizar el beneficio esperado a toda costa sin importar el número de épocas. De hecho algunos algoritmos obtienen resultados muy cercanos con un números de épocas muy inferior. El gráfico 4.1 se muestra la recompensa esperada en cada una de las épocas junto a una linea de suavizado para apreciar mejor su tendencia. Se puede observar que la recompensa obtenida finalmente no es la máxima que se ha obtenido a lo largo del proceso, esto es debido a la acción (en mal momento) de las políticas greedy.

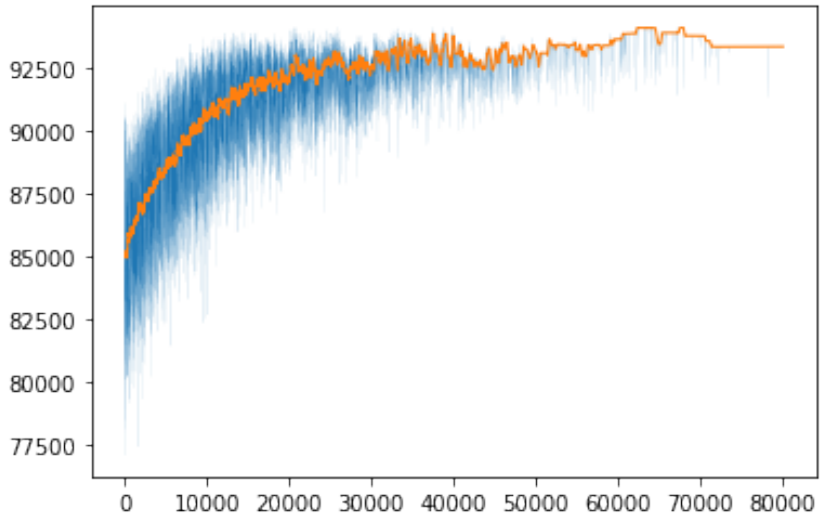

Figura 4.1: Entrenamiento del algoritmo SARSA (épocas y recompensa obtenida)

Como estamos considerando la demanda determinista, la fijación de precios en un mes supondrá siempre una demanda fija, por lo tanto podemos obtener todos los precios que establecen nuestros algoritmos a lo largo del proceso.

Algoritmo	Mes1	Mes2	Mes3	Mes4	Mes5	Mes6	Mes7	Mes8	Mes9	Mes10	Mes11	Mes12
Q-Learning	925	925	925	925	925	925	925	925	925	975	975	975
SARSA	850	925	900	925	850	950	975	925	925	950	925	975
Doble Q-Learning	975	850	975	850	925	975	975	850	975	975	975	850
Expected SARSA	900	800	900	900	900	900	900	900	975	975	975	975

Cuadro 4.3: Precios fijados para el producto A (en u.m.)

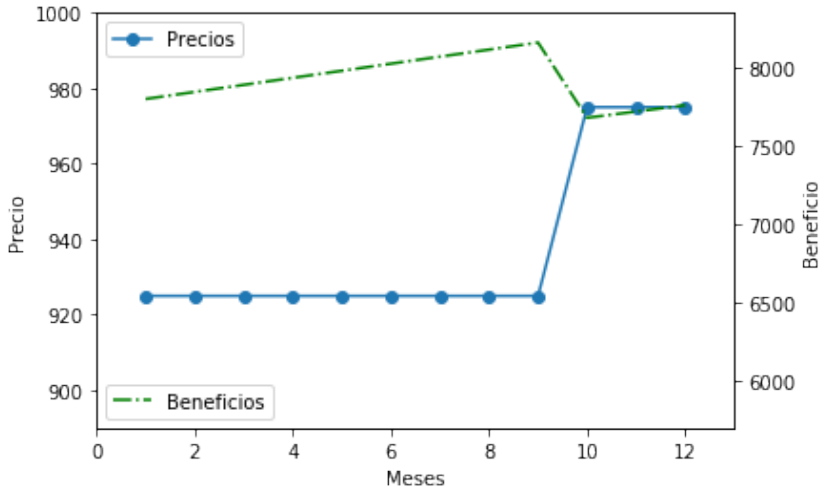

Figura 4.2: Precios fijados por Q-learning y beneficio asociado.

Ahora vamos a evaluar el modelo considerando la demanda estocástica, por lo tanto siguiendo la política óptima de cada algoritmo, no se obtienen siempre los mismos resultados. Así que para medir cada una de las políticas calculamos la media y la desviación típica de un total de 100 repeticiones.

Algoritmo	Media de las recompensas obtenidas	Desviación de las recompensas obtenidas
Q-Learning	89.528,33	3.746,73
SARSA	88.688,89	4.878,41
Doble Q-Learning	89.138,67	5.552,68
Expected SARSA	86.030,7	5.223,50

Cuadro 4.4: Resultados obtenidos con el uso de demanda determinista (en u.m.).

Volvemos a ver resultados muy semejantes entre los algoritmos luego no se pueden sacar grandes conclusiones entre ellos. Parece que la política óptima del Expected SARSA es la que peor se ha adaptado a esta demanda estocástica. Lógicamente la recompensa obtenida ha disminuido con la teórica del modelo determinista ya que la política no ha sido entrenada para adaptarse al caso estocástico.

Veamos que ocurre si entrenamos directamente los modelos con esta demanda estocástica.

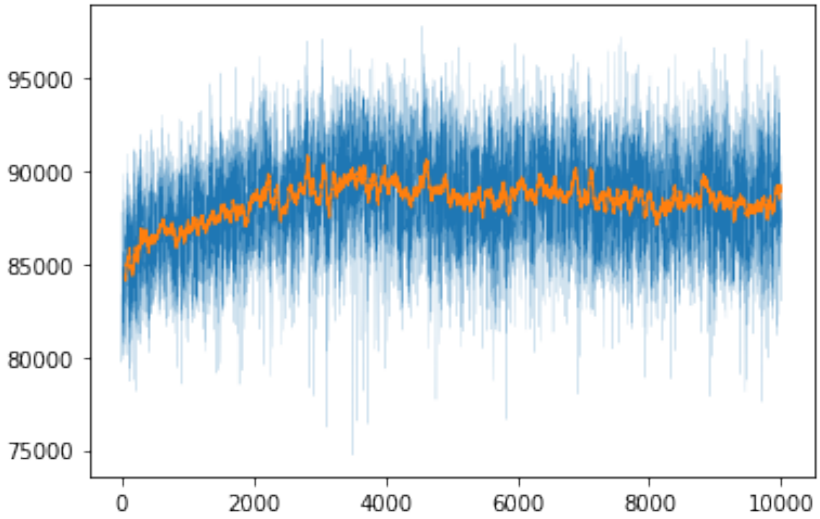

Figura 4.3: Entrenamiento del algoritmo Doble Q-Learning (épocas y recompensa obtenida).

Algoritmo	Media de las recompensas obtenidas	Desviación de las recompensas obtenidas
Q-Learning	89.232,9	2.678,60
SARSA	90.215	1.556,58
Doble Q-Learning	89.317	2.415,77
Expected SARSA	86.795,15	2.752,96

Cuadro 4.5: Resultados obtenidos con el uso de demanda estocástica (en u.m.).

En la gráfica 4.3 se puede apreciar que pese a que disminuya el ε a lo largo de las épocas, las variaciones entre las recompensas se mantienen hasta el final del proceso, la componente del error de la demanda provoca que tomando la mismas decisiones no siempre se obtengan las mismas recompensas. Nos quedan unos resultados bastante comparables a los anteriores, aunque se observa que se ha logrado reducir la desviación de las recompensas con esta adaptación a la propia aleatoriedad del modelo. Además el algoritmo SARSA si que ha mejorado respecto a su entrenamiento con demanda determinista. Por lo tanto podemos decir que ante esta aleatoriedad simétrica y aditiva resulta más beneficioso el entrenamiento con demanda estocástica.

4.1.2. Producto B.

Esta vez ajustamos los cuatro algoritmos para maximizar el beneficio anual obtenido al vender el producto B. El espacio discreto de precios que hemos tomado va de $p = 350$ u.m. a $p = 450$ u.m. de 10 en 10, así que tenemos nuevamente 11 posibles precios a elegir en cada mes. Entrenado los algoritmos con la función de demanda determinista obtenemos los siguientes resultados:

	Recompensa obtenida	Meses empleados	Épocas
Q-Learning	297.360	12	20.000
SARSA	296.836	12	40.000
Doble Q-Learning	293.528	12	20.000
Expected SARSA	294.476	12	40.000

Cuadro 4.6: Resultados obtenidos para el producto B.

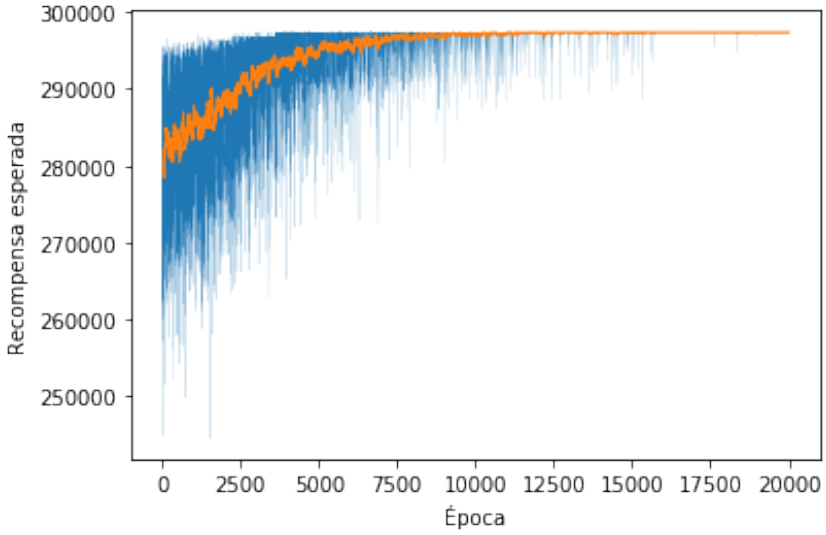

Figura 4.4: Entrenamiento del algoritmo Q-Learning.

En este producto sí que observamos diferencia entre los resultados de los cuatro algoritmos, parece que Q-Learning y SARSA son los que mayor recompensa esperada generan a través de sus políticas óptimas. Observar que han sido necesarias más épocas en los algoritmos SARSA y Expected SARSA. Nuevamente el algoritmo que más recompensa esperada genera es el Q-Learning, como se puede ver en la gráfica 4.4, alcanza rapidamente un óptimo que puede ser global.

Algoritmo	Mes1	Mes2	Mes3	Mes4	Mes5	Mes6	Mes7	Mes8	Mes9	Mes10	Mes11	Mes12
Q-Learning	400	390	410	390	380	380	370	410	370	400	370	370
SARSA	370	380	410	400	370	390	400	380	420	380	380	360
Doble Q-Learning	380	400	440	400	390	450	440	350	350	350	380	350
Expected SARSA	350	350	370	360	400	410	440	440	410	420	380	350

Cuadro 4.7: Precios fijados para el producto B (en u.m.).

Podemos comparar estas fijaciones de precios con una fijación más estandar por parte del negocio, un precio fijo durante todos los meses. Sin embargo, la demanda no tiene por qué ser la misma todos los meses ni el vendedor tiene por qué poder conocer sus posibles ventas de forma determinista, por lo que el problema es en general más complicado a la hora de ajustar un precio fijo que en nuestro caso.

A pesar de ello, si el vendedor fija un precio de 380 todos los meses obtendrá un beneficio de 293,908 u.m. y si fija un precio de 390 obtendrá un beneficio de 293,580 u.m. Por tanto, los tres mejores algoritmos mejoran ampliamente estas políticas de precio fijo, pese a tener toda la información de la demanda disponible a la hora de determinar el precio fijo.

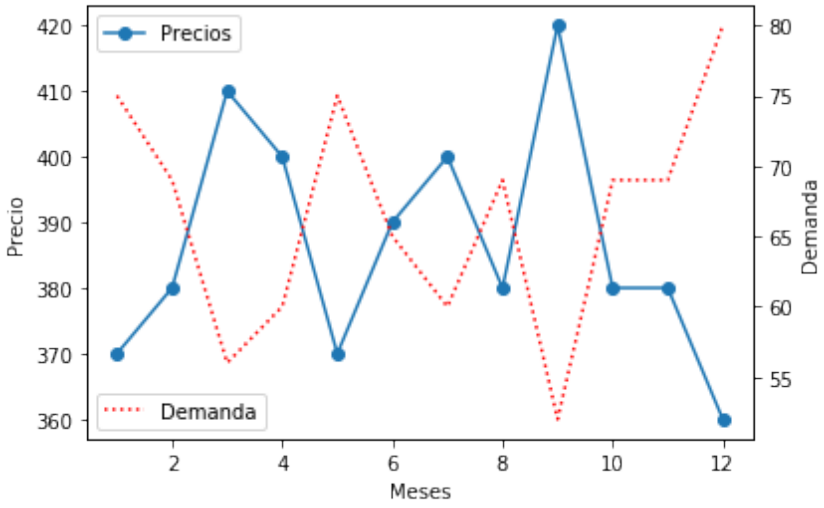

Figura 4.5: Precios fijados por SARSA y demanda asociada.

Si analizamos los precios fijados por el algoritmo SARSA, en los últimos meses se producen bajadas de precios para acabar con el stock antes del final del proceso. De esta fijación de precios destaca la variabilidad de los precios mes a mes pese a que la demanda es la misma en cada uno de ellos.

Observamos ahora los resultados de evaluar los modelos teniendo en cuenta la demanda estocástica correspondiente.

Algoritmo	Media de las recompensas obtenidas	Desviación de las recompensas obtenidas
Q-Learning	284.391,88	8.719,48
SARSA	283.557,12	8.908,04
Doble Q-Learning	284.871,84	8.176,62
Expected SARSA	276.098,92	6.649,42

Cuadro 4.8: Resultados obtenidos con el uso de demanda determinista (en u.m.).

Notar que el algoritmo que mejores resultados ha obtenido es el Doble Q-Learning pese a que sus valores en el entrenamiento eran los peores, por tanto se podría decir que la política generada por el Doble Q-Learning se ha ajustado bien a la demanda estocástica. El que peores resultados ha obtenido es el Expected SARSA.

Veamos que ocurre si entrenamos directamente los modelos con esta demanda estocástica.

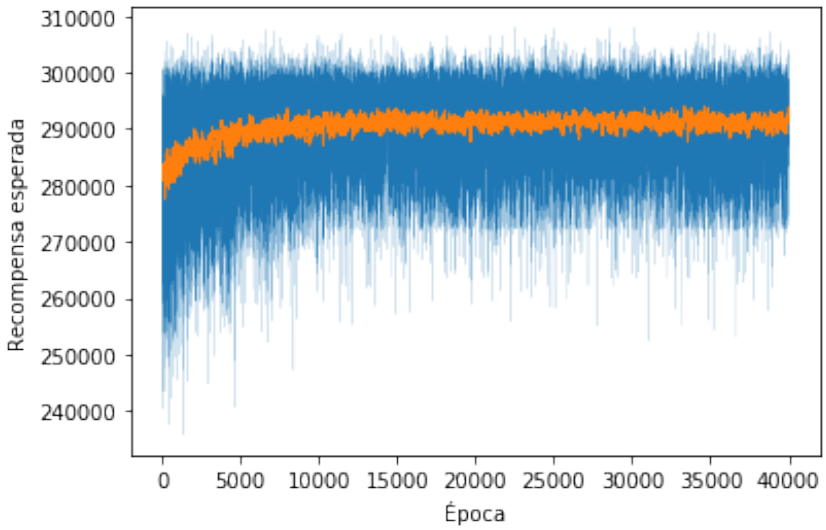

Figura 4.6: Entrenamiento del algoritmo Expected SARSA.

Algoritmo	Media de las recompensas obtenidas	Desviación de las recompensas obtenidas
Q-Learning	287.36,88	7.985.15
SARSA	291.960,48	6.964,92
Doble Q-Learning	287.814,72	4.276,05
Expected SARSA	290.971,08	7.547,90

Cuadro 4.9: Resultados obtenidos con el uso de demanda estocástica (en u.m.).

Observamos que todos los algoritmos mejoran sus resultados entrenando directamente con la demanda estocástica y la desviación de estas repeticiones se reduce. El que mayor beneficio medio obtiene es el Expected SARSA, que es curiosamente el que peor resultados obtenía al ser entrenado con la demanda determinista. Esto muestra que las polítcas óptimas generadas por medio de demandas deterministas y estocásticas poco tienen que ver. Nuevamente estamos ante un término del error simétrico (esta vez con demanda multiplicativa), lo que hace pensar que esta simetría favorece al aprendizaje con demanda estocástica.

4.1.3. Producto C.

Finalmente ajustamos los cuatro algoritmos para maximizar el beneficio anual obtenido al vender el producto C. En este caso el espacio de posibles precios a fijar en cada mes es

$$p \in [340, 350, 360, 370, 375, 380, 390, 400, 405, 410, 420].$$

Entrenado los algoritmos con la función de demanda determinista obtenemos los siguientes resultados:

	Recompensa obtenida	Meses empleados	Épocas
Q-Learning	77.880	12	15.000
SARSA	76.660	12	15.000
Doble Q-Learning	60.815	12	15.000
Expected SARSA	77.005	12	15.000

Cuadro 4.10: Resultados obtenidos para el producto C.

Hemos realizado el mismo número de épocas en todos los algoritmos para poder comparar sus velocidades de convergencia a políticas óptimas.

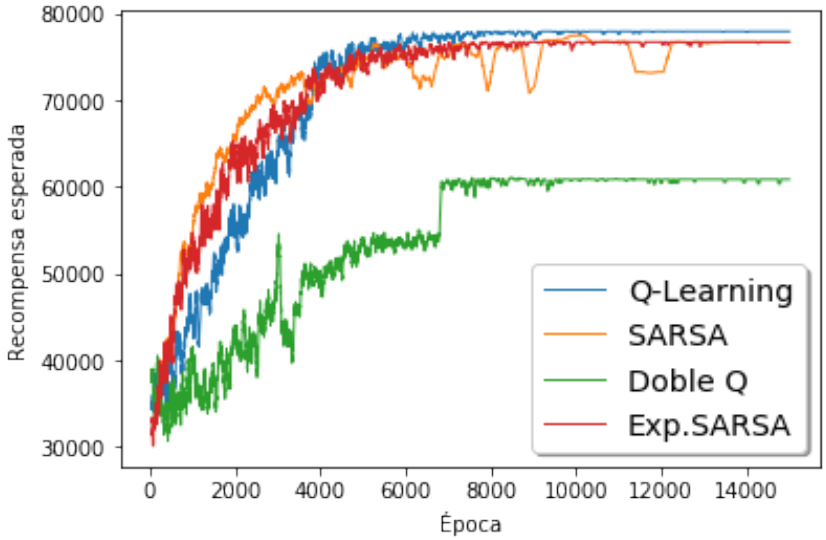

Figura 4.7: Entrenamiento con demanda determinista.

Observamos que los algoritmos SARSA, Q-Learning y Expected SARSA obtienen los mejores resultados. Mientras que SARSA y Expected SARSA consiguen mejorar al Q-Learning al principio de las épocas, el Q-learning acaba mejorando al resto a partir de la época 4000 aproximadamente. Es bastante destacable el pobre resultado del Doble Q-Learning, esto es debido a que al basarse en dos funciones de valor-acción Q_1 y Q_2, no le ha dado tiempo a encontrar una política suficientemente buena y se ha quedado a mitad de su aprendizaje. Por lo tanto serían necesarias más épocas para optimizar el Doble Q-Learning.

Algoritmo	Media de las recompensas obtenidas	Desviación de las recompensas obtenidas
Q-Learning	63.020	19.030,60
SARSA	69.225,05	17.082,55
Doble Q-Learning	49.777,45	23.749,89
Expected SARSA	59.915,7	22.452,59

Cuadro 4.11: Resultados obtenidos con demanda determinista.

Los resultados obtenidos al evaluar las políticas óptimas de cada algoritmo nos permiten ver que la política generada por el SARSA es la que mejor se ajusta. Lógicamente la política generada por el Doble Q-Learning es la que peores resultados obtiene. Se puede observar la gran desviación de los recompensas debida al término del error elegido.

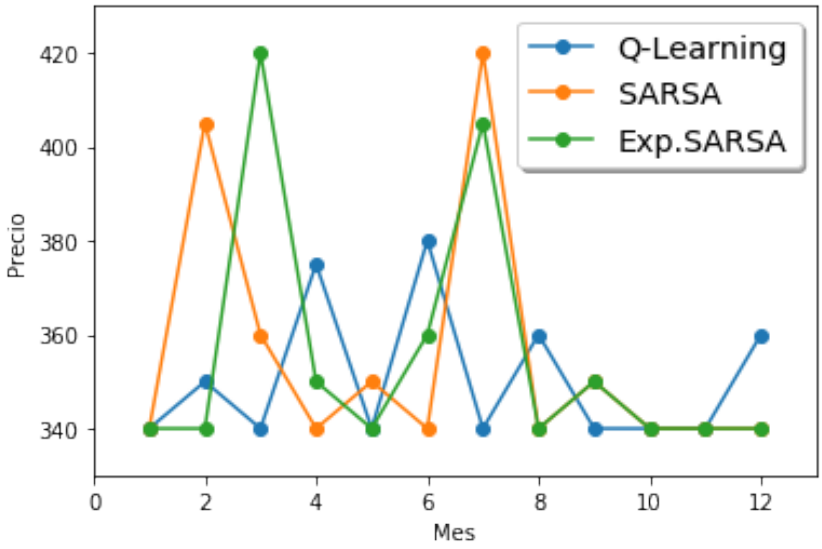

Figura 4.8: Precios fijados por los tres mejores algoritmos.

Destacar que la fijación de precios de los algoritmos SARSA y Expected SARSA es muy similar. Ambos establecen precios bajos en todos los meses menos en un par, donde establecen precios muy altos, el Q-learning en cambio va fijando precios medios con más frecuencia.

Entrenando los algoritmos con la demanda estocástica se obtienen los siguientes resultados:

Algoritmo	Media de las recompensas obtenidas	Desviación de las recompensas obtenidas
Q-Learning	47.381,3	19.554,72
SARSA	58.011,5	20.199,84
Doble Q-Learning	35.556,95	20.376,20
Expected SARSA	43.162,55	21.853,09

Cuadro 4.12: Resultados obtenidos con demanda estocástica.

Cabe destacar que el método que mejores resultados obtiene es el SARSA con bastante diferencia, seguido del Q-Learning. Añadir que en comparación con los modelos obtenidos con la demanda determinista, todos los beneficios medios se han reducido, además la desviación de estas recompensas no ha disminuido significativamente. Por lo tanto parece que en el caso de esta demanda estocástica es mejor entrenar los modelos con demanda determinista. Podemos encontrarle una justificación a estos resultados, se trata de un término del error muy asimétrico y además aparece en la demanda de forma multiplicativa por lo tanto genera un comportamiento muy impredecible y bastante difícil de aprender para nuestros algoritmos.

4.1.4. Conclusiones

Con estos resultados hemos analizado la variación de los distintos algoritmos aplicados a estos problemas de fijación de precios. Hemos visto que no siempre es el mismo método el que mejor funciona, depende mucho del tipo de problema en el que estamos y de los parámetros del mismo. Esta similitud de resultados hace pensar que para estos problemas en concreto sea suficiente con un algoritmo Q-Learning y un algoritmo SARSA dado que su implementación es más sencilla. No obstante, parece que todos los algoritmos nos pueden proporcionar resultados aceptables.

Hemos podido observar que dependiendo del problema merece la pena entrenar el modelo con la demanda estocástica o con la demanda determinista. En el producto A y en el B hemos visto que en cierta manera es mejor entrenar los algoritmos con demanda estocástica, mientras que en el producto C

parece que lo mejor es entrenarlos con la demanda determinista. Estas apreciaciones están directamente relacionadas con el término del error de la demanda, si tenemos un error simétrico merece la pena entrenar los algoritmos con demanda estocástica, mientras que si el término del error es muy asimétrico no compensa usar la demanda estocástica en el entrenamiento.

Destacar que la implentación directa del problema en *Python*, sin la ayuda de ninguna librería específica, nos ha permitido la creación del entorno del problema con total flexibilidad (las recompensas, probabilidades de transición, etc). Además en el aspecto de los algoritmos nos ha permitidos variar todos los parámetros con total libertad.

4.2. Aprendizaje con dos agentes

En este caso vamos a simular un problema ficticio que se basa en la estructura del anterior pero añadiéndole algunas modificaciones con el fin de que sirva de ejemplo práctico del aprendizaje por refuerzo multi-agente. Vamos a desarrollar el problema con dos agentes que van a corresponder con dos negocios dedicados a la venta del mismo producto, esa competencia hará que los precios de uno influyan en cierta manera en las ventas del otro.

En primer lugar vamos a suponer que queremos maximizar el beneficio a lo largo de los 12 meses de un negocio más pequeño que su principal competidor, se supone que el producto a vender es el producto A y que por tanto se tienen 105 piezas en stock al inicio del proceso de fijación de precios y el coste de almacenamiento es $h = 5$ u.m. por unidad y mes. En este caso los posibles precios a fijar al inicio de cada mes son los siguientes: $p \in [725, 775, 825, 875, 925, 975]$, y la demanda asociada a este producto en este negocio viene directamente determinada por los propios precios establecidos y por los que fija el competidor. Si consideramos las filas como el precio fijado por nuestro agente (en orden ascendiente) y las columnas el precio del competidor, la matriz de demanda mensual es la siguiente:

$$D_1 = \begin{pmatrix} 10 & 11 & 12 & 13 & 14 & 15 \\ 9 & 10 & 11 & 12 & 13 & 14 \\ 8 & 9 & 10 & 11 & 12 & 13 \\ 7 & 8 & 9 & 10 & 11 & 12 \\ 6 & 7 & 8 & 9 & 10 & 11 \\ 5 & 6 & 7 & 8 & 9 & 10 \end{pmatrix}$$

Respecto al comportamiento del competidor vamos a suponer que, como es tan superior en volumen de ventas, los precios fijados por nuestro agente no influyen sobre sus demandas. Una vez considerado esto, se ha observado que sus precios siguen el siguiente comportamiento:

- Los primeros 3 meses toman $p = 925$ o $p = 975$.
- Los 3 siguientes meses los posibles precios son $p = 875$ y $p = 925$.
- Entre el mes 7 y el mes 9 fijan $p = 825$, $p = 875$ o $p = 925$.
- Los últimos 3 meses los precios pueden ser más variados, $p = 725, 775, 825, 875$.

Si consideramos que cada uno de los posibles precios en cada mes tiene la misma probabilidad de ser fijado, podemos determinar una política estacionaria para el oponente y aplicar a nuestro agente el algoritmo de Opponent Modelling con el fin de adaptarse lo mejor posible a esta política de la competencia.

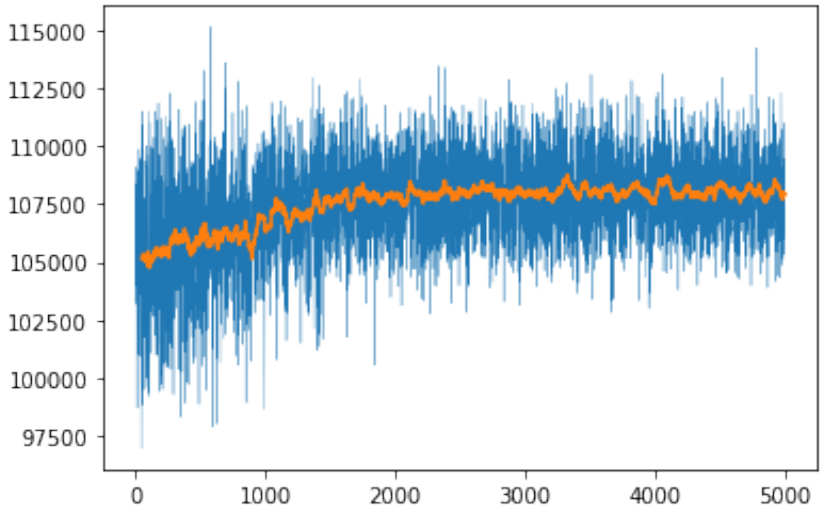

Figura 4.9: Entrenamiento con Opponent Modelling.

Después de realizar 5000 épocas, se ve en en la gráfica 4.9 que el beneficio va aumentando lentamente a través de ellas, aunque parece que a partir de la época 2000 no avanza mucho. Si seguimos la política óptima generada por este método, el beneficio medio obtenido tras 100 repeticiones es de 106.583,44 u.m. con una desviación de 2.509,58 u.m. La fluctuación de las recompensas obtenidas por nuestro agente época a época se debe a la aleatoriedad en la fijación de precios del oponente.

Ahora vamos a cambiar de problema, y vamos a suponer que los precios sí influyen en el oponente. Consideramos que ambos productos se reparten totalmente una demanda constante en cada periodo. Así que el agente 2 tiene una matriz de demandas $D_2 = D_1^t$. Con el fin de adaptar el problema a una interacción de ambos agentes en tiempo real, vamos a considerar los periodos de 1 hora, el stock inicial de 210 piezas y el límite para venderlas 24 horas. Además los posibles precios son los mismos que antes $p \in [725, 775, 825, 875, 925, 975]$.

Vamos a ver como se enfrentan dos políticas no estacionarias en el algoritmo de Opponent Modelling, diseñado para trabajar con una política estacionaria. Suponemos que los dos agentes aprenden a lo largo de las épocas (en este caso días) observando los precios fijados por el oponente y siguiendo este algoritmo.

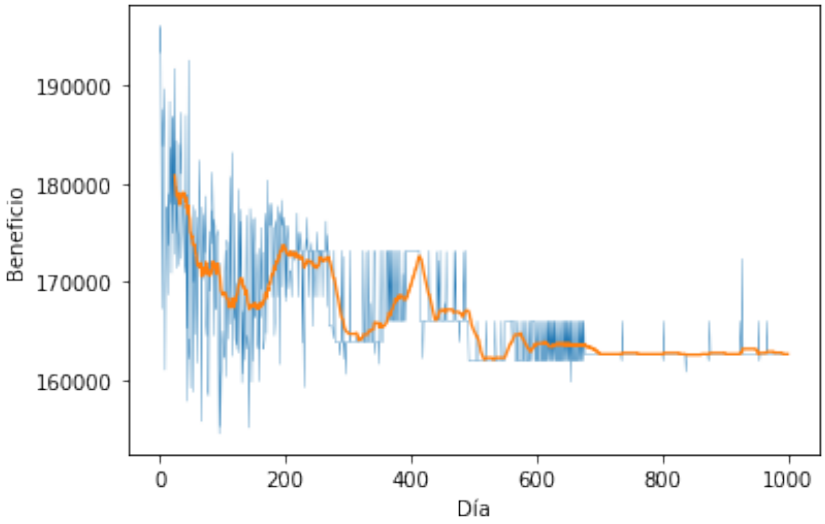

Figura 4.10: Beneficios del Agente 1 siguiendo Opponent modelling.

Esta vez hemos establecido un proceso de aprendizaje de 1000 días, el Agente 1 tiene beneficio el ultimo día 162.680 u.m., muy inferior a los obtenidos los primeros días donde tenía 180.000 aproximadamente.

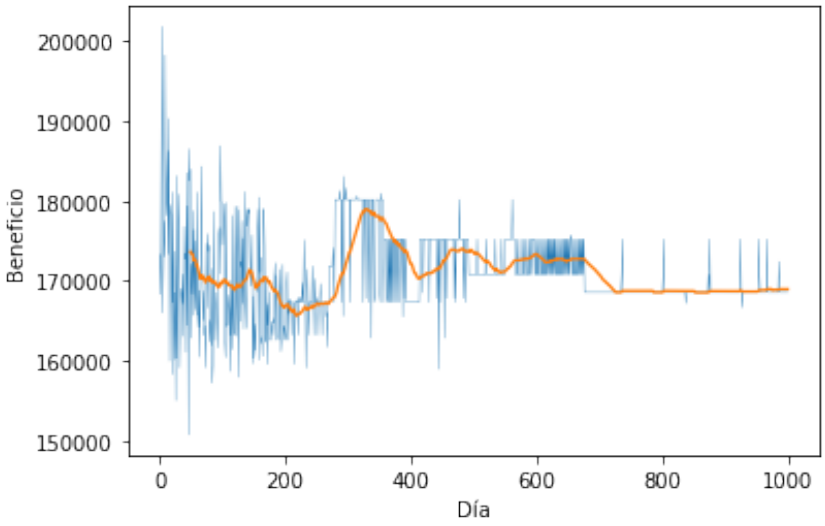

Figura 4.11: Beneficios de Agente 2 siguiendo Opponent modelling.

El Agente 2 tiene un beneficio el ultimo día de 168.570 u.m., un poco inferior a lo que obtiene los primeros días. Observamos que se alcanza una especie de equilibrio entre ambos agentes llegando a tener un beneficio relativamente similar. Sin embargo, vemos que el algoritmo no funciona muy bien en cuanto a la idea de maximizar la recompensa, pues al comienzo del proceso el beneficio de ambos agentes era mayor y ha ido decreciendo a lo largo de los días (épocas). Por tanto se podría decir que ambos han sido perjudicados por esta estrategia, sus políticas iniciales les generaban más beneficios.

4.2.1. Conclusiones

Pese a que estamos ante un problema ficticio y aparentemente no muy complejo, nos es suficiente para observar las ventajas e inconvenientes de los métodos de mejor respuesta, en particular del algoritmo Opponent Modelling.

Estos resultados confirman que el funcionamiento del algoritmo de Opponent Modelling se basa en la política estacionaria del oponente. Primero hemos podido ver como se adaptaba nuestro agente a una política estacionaria (aunque tuviera parte aleatoria) maximizando el beneficio. La política que ha generado este entrenamiento trata de sacar máximo provecho a la política del oponente maximizando las recompensas propias.

En el segundo problema analizado, al suponer que ambos agentes aprenden al mismo tiempo, el algoritmo no ha funcionado bien y no ha dado buenos resultados para ninguno de los dos agentes. El algoritmo no ha podido adaptarse a ninguna política del oponente ya que esta también cambiaba continuamente.

Bibliografía

[1] I. AREL, C. LIU, T. URBANIK Y A.G. KOHLS, *Reinforcement learning-based multi-agent system for network traffic signal control*, IET Intelligent Transport Systems, 2010.

[2] E.ARKAN Y W.JAMMERNEGG, *The newsvendor problem with a general price dependent demand distribution*, University of Viena, 2009.

[3] M.BOWLING Y M.VELOSO, *Rational and Convergent Learning in Stochastic Games*, Proceedings of the 17th international joint conference on Artificial intelligence, Volume 2, 1021–1026, 2001.

[4] W.CHEN, H.LIU Y D.XU, *Dynamic Pricing Strategies for Perishable Product in a Competitive Multi-Agent Retailers Market*, Journal of Artificial Societies and Social Simulation 21(2) 12, 2018.

[5] V.L.R.CHINTHALAPATI, N.YADATI Y R.KARUMANCHI, *Learning Dynamic Prices in MultiSeller Electronic Retail Markets With Price Sensitive Customers, Stochastic Demands, and Inventory Replenishments*, IEEE Transactions on systems, 2006.

[6] A.V. DEN BOER, *Dynamic Pricing and Learning*, PhD thesis, University of Amsterdam, 2015.

[7] FERRARA, MONICA, *A reinforcement learning approach to dynamic pricing*, Tésis, Politecnico de Torino, 2018.

[8] J.FILAR Y K.VRIEZE, *Competitive Markov Decission Process*, Springer-Verlag, 1997.

[9] GARCÍA-CALDERÓN CHAVEZ, SAUL ABRAHAM, *Pricing Recommendation by Applying Statistical Modeling Techniques*, Facultad De Informática DE Barcelona UPC, 2017.

[10] P.HARSHA, R.NATARAJAN Y D.SUBRAMANIAN, *A prescriptive machine learning framework to the price-setting newsvendor problem*, 2018.

[11] J. HU Y M.P. WELLMAN, *Multiagent Reinforcement Learning: Theoretical Framework and an Algorithm*, Proceedings of the Fifteenth International Conference on Machine Learning, 242–250, 1998.

[12] J.HUANG, M.LENG Y M.PARLAR, *Demand Functions in Decision Modeling: A Comprehensive Survey and Research Directions*, Decision Sciences Journal, Volume 44 Number 3, 2013.

[13] J.JIN, C.SONG , H.LI, K.GAI, J.WANG Y W.ZHANG, *Real-Time Bidding with Multi-Agent Reinforcement Learning in Display Advertising*, Proceedings of the Tenth ACM International Conference on Web Search and Data Mining, 661–670, 2017.

[14] J. W. LEE, J. PARK, JANGMIN O., J. LEE, AND E. HONG, *A Multiagent Approach to Q-Learning for Daily Stock Trading* IEEE Transactions on systems, man, and cybernetics, 2007.

[15] J.LIU, Y.ZHANG , X.WANG, Y.DENG, X.WU Y M.XIE *Dynamic Pricing on e-commerce platform with deep reinforcement learning*, ICLR, 2018, `https://openreview.net/pdf?id=HJMRvsAcK7` (visitado el 23-11-2019).

[16] R. MAESTRE, J. DUQUE, A. RUBIO Y J. AREVALO, *Reinforcement Learning for Fair Dynamic Pricing*, Intelligent Systems Conference, 2018.

[17] 0.L. MANGASARIAN AND H. STONE, *Two-Person Nonzero-Sum Games and Quadratic Programming*, Journal of Mathematical Analysis and Applications **9**, 348-355, 1964.

[18] NATH, SWAPRAVA *Nash Theorem and its proof*, Notas del curso cs711, IIT Kanpur, `https://www.cse.iitk.ac.in/users/swaprava/courses/cs711/nash-proof.pdf` (visitado el 16-11-2019).

[19] NETO, GONZALO, *From Single-Agent to Multi-Agent Reinforcement Learning: Foundational Concepts and Methods*, Learning theory course, 2005.

[20] J. PÉREZ, J. L. JIMENO Y E. CERDÁ, *Teoría de Juegos*, Pearson, 2004.

[21] PYTHON SOFTWARE FOUNDATION *Python Language Reference, version 3.0.*, disponible en `http://www.python.org`

[22] R. RANA Y F. OLIVEIRA, *Real-Time Dynamic Pricing in a Non-Stationary Environment using Model-Free Reinforcement Learning*, Loughborough University y ESSEC University, 2014.

[23] D.SILVER, A.HUANG, C.J. MADDISON, A.GUEZ, L.SIFRE, G.VAN DEN DRIESSCHE, J.SCHRITTWIESER, I.ANTONOGLOU, V.PANNEERSHELVAM, M.LANCTOT, S.DIELEMAN, D.GREWE, J.NHAM, N.KALCHBRENNER, I.SUTSKEVER, T.LILLICRAP, M.LEACH, K.KAVUKCUOGLU, T.GRAEPEL Y D.HASSABIS , *Mastering the game of Go with deep neural networks and tree search*, Nature volume 529, (484–489), 2016.

[24] SINGH, SATINDER P., T.JAAKKOLA, M. L. LITTMAN Y C. SZEPESVARI, *Convergence results for single-step on-policy rein- forcement learning algorithms*, Machine Learning **38**, 287–308, 2000.

[25] S.SHAKYA, F.OLIVEIRA Y G.OWUSU, *An Application of EDA and GA to Dynamic Pricing*, Proceedings of GECCO 2007: Genetic and Evolutionary Computation Conference, 585–592, 2007.

[26] SUTTON, RICHARD, *Learning to Predict by the Methods of Temporal Differences*, Machine Learning **3**, 9–44, 1998.

[27] R. SUTTON Y A. BARTO, *Reinforcement Learning: An Introduction*, The MIT Press, 1998.

[28] SZEPESVARI, CSABA, *Algorithms for Reinforcement Learning*, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers, 2009.

[29] TESAURO, GERALD, *Extending Q-Learning to General Adaptive Multi-Agent Systems*, IBM Thomas J. Watson Research Center, `https://papers.nips.cc/paper/2503-extending-q-learning-to-general-adaptive-multi-agent-systems.pdf` (visitado el 23-11-2019).

[30] WATKINS, CHRISTOPHER J. C. H., *Learning from Delayed Rewards*, PhD thesis, Cambridge University,1989.

[31] C. J. C. H. WATKINS Y P.DAYAN, *Technical note: Q- learning*, Machine Learning **8**, 279-292, 1992.

Anexo

En el siguiente anexo se muestra el código de *Python* necesario para la implementación del problema y la ejecución de los algoritmos.

Entorno del problema

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import random
#Definición de las demandas
def demand1(x):
return 18-0.01*x
def demand2(x):
return 1000*np.exp(-0.007*x)
def demand3(x):
return 750/((x-325)**(1.2))
#Demandas estocásticas
def demand1E(x):
return 18-0.01*x +round(1.5*np.random.normal())
def demand2E(x):
return 1000*np.exp(-0.007*x)*((np.random.normal()/8+1))
def demand3E(x):
return 750*np.random.exponential(1)/((x-325)**(1.2))
#Posibles precios
dicactions1={0:725,1:750,2:775,3:800,4:825,5:850,6:875,7:900,8:925,9:950,10:975}
dicactions2={0:350,1:360,2:370,3:380,4:390,5:400,6:410,7:420,8:430,9:440,10:450}
dicactions3={0:340,1:350,2:360,3:370,4:375,5:380,6:390,7:400,8:405,9:410,10:420}
dicactions=dicactions1
demand=demand1E
n_piezas=105  #numero de piezas en stock
n_periodos=12 #numero de meses para vender las piezas
h=5 #coste de almacenamiento

#Creación del entorno
class Env():
def __init__(self):
self.size = n_piezas+1;
self.periods=n_periodos
self.posX = [n_piezas,n_periodos];
self.endX = 0;
self.actions = [0,1,2,3,4,5,6,7,8,9,10];
self.stateCount = self.size;
self.actionCount = len(self.actions);

def reset(self):
```

```
self.posX = [n_piezas,n_periodos];
self.done = False;
return [n_piezas,n_periodos],0, False;

# take action
def step(self, action):
nextState = [max(self.posX[0]-int(demand(dicactions[action])),0),self.posX[1]-1];

done = (nextState[1]==self.endX or nextState[0]==self.endX);

reward = (dicactions[action])*(self.posX[0]-nextState[0])-h*self.posX[0];

return nextState, reward, done;

# return a random action
def randomAction(self):
return np.random.choice(self.actions);

def policyFunction(self,qtable,epsilon):
Action_probabilities = np.ones(self.actionCount, dtype = float)
 * epsilon / self.actionCount
best_action = np.argmax(qtable[self.posX[1]][self.posX[0]])
Action_probabilities[best_action] += (1.0 - epsilon)
return Action_probabilities
```

Algoritmo Q-Learning

```
env = Env()
qtable = np.zeros([n_periodos+1,env.stateCount, env.actionCount]).tolist()

# hyperparameters
epochs = 15000
epsilon = 0.8
decay = epsilon/epochs
#stats
episode_lengths=[0]*epochs
episode_rewards=[0]*epochs
# training loop
for i in range(epochs):
state, reward, done = env.reset()
steps=0
while not done:
# count steps to finish game
steps+= 1
action_probabilities = env.policyFunction(qtable,epsilon)
action = np.random.choice(np.arange(len(action_probabilities)),
 p = action_probabilities)
# take action
next_state, reward, done = env.step(action)
env.posX=next_state
# Update statistics
episode_rewards[i] += reward
# update qtable
qtable[state[1]][state[0]][action] = (1-0.5)* qtable[state[1]][state[0]][action]
+0.5*(reward +  max(qtable[state[1]-1][next_state[0]]))#*gamma
```

```
# update state
state = next_state
# update epsilon
epsilon = epsilon-decay*epsilon
episode_lengths[i] = steps
print("epoch #", i+1, "/", epochs)
print("\nDone in", steps, "steps".format(steps))
# test con política óptima
profit=[0]*100
for i in range(100):
state, reward, done = env.reset()
steps=0
step_reward=[0.0]*13
actions=[0]*13
R=0
while not done:
steps+= 1
action_probabilities = env.policyFunction(qtable,0)
action = np.random.choice(np.arange(len(action_probabilities)),
 p = action_probabilities)
actions[steps]=action
# take action
next_state, reward, done = env.step(action)
env.posX=next_state

# Update statistics
step_reward[steps] = reward
R=R+reward
# update state
state = next_state
profit[i]=R
print(actions)
print(profit)
#calculo de la media
sum=0
for i in range(100):
sum=sum+profit[i]
mean=sum/100
#calculo de la desviación
dt=0
for i in range(100):
dt=(mean-profit[i])**2+dt
dtf=np.sqrt(dt/100)
print(mean)
print(dtf)
```

Algoritmo SARSA

```
env = Env()
epsilon = 0.8
total_episodes = 25000
decay = epsilon/total_episodes
lr_rate = 0.5
gamma = 1
Q = np.zeros((13,env.stateCount, env.actionCount))
```

```
episode_rewards=[0]*total_episodes
def choose_action(state):
action=0
if np.random.uniform(0, 1) < epsilon:
action = env.randomAction()
else:
action = np.argmax(Q[state[1],state[0], :])
return action

def learn(state, state2, reward, action, action2):
predict = Q[state[1],state[0], action]
target = reward + gamma * Q[state2[1],state2[0], action2]
Q[state[1],state[0], action] = Q[state[1],state[0], action] +
 lr_rate * (target-predict)
return Q

# Start
rewards=0

for episode in range(total_episodes):
t = 0
state,reward,done =env.reset()
#choose action
action_probabilities = env.policyFunction(Q,epsilon)
action = np.random.choice(np.arange(len(action_probabilities)),
 p = action_probabilities)

while not done:
nextstate, reward, done = env.step(action)
env.posX=nextstate
action2_probabilities = env.policyFunction(Q,epsilon)
action2 = np.random.choice(np.arange(len(action2_probabilities)),
 p = action2_probabilities)
learn(state, nextstate, reward, action, action2)
state = nextstate
action = action2
t += 1
rewards+=1
episode_rewards[episode] += reward
epsilon=epsilon-decay*epsilon


# test con política óptima
profit=[0]*100
for i in range(100):
state, reward, done = env.reset()
steps=0
step_reward=[0.0]*13
actions=[0]*13
R=0
while not done:

steps+= 1
action_probabilities = env.policyFunction(Q,0)
action = np.random.choice(np.arange(len(action_probabilities)),
 p = action_probabilities)
actions[steps]=action
# take action
```

```
next_state, reward, done = env.step(action)
env.posX=next_state

# Update statistics
step_reward[steps] = reward
R=R+reward
# update state
state = next_state
profit[i]=R

print(actions)
print(profit)
#cálculo de la media
sum=0
for i in range(100):
sum=sum+profit[i]
mean=sum/100
#cálculo de la desviación
dt=0
for i in range(100):
dt=(mean-profit[i])**2+dt
dtf=np.sqrt(dt/100)
print(mean)
print(dtf)
```

Algoritmo Doble Q-Learning

```
env = Env()
qtable1 = np.zeros([n_periodos+1,env.stateCount, env.actionCount]).tolist()
qtable2 = np.zeros([n_periodos+1,env.stateCount, env.actionCount]).tolist()
# hyperparameters
epochs = 40000
gamma = 1
epsilon = 0.8
decay = epsilon/epochs
alpha=1
#stats
episode_lengths=[0]*epochs
episode_rewards=[0]*epochs
for i in range(epochs):
state, reward, done = env.reset()
steps=0
while not done:
steps+= 1
action_probabilities = env.policyFunction(qtable1+qtable2,epsilon)
action = np.random.choice(np.arange(len(action_probabilities)),
 p = action_probabilities)
# take action
next_state, reward, done = env.step(action)
env.posX=next_state
# Update statistics
episode_rewards[i] += reward
# update qtable value with Bellman equation
num_ale=np.random.rand()
if(num_ale<0.5):
```

```
qtable1[state[1]][state[0]][action] += alpha*(reward +
 qtable2[state[1]-1][next_state[0]][np.argmax(qtable1[state[1]-1][nextstate[0]])]
 -qtable1[state[1]][state[0]][action])#*gamma
else:
qtable2[state[1]][state[0]][action] += alpha*(reward +
  qtable1[state[1]-1][next_state[0]][np.argmax(qtable2[state[1]-1][nextstate[0]])]
  -qtable2[state[1]][state[0]][action])#*gamma
# update state
state = next_state
epsilon = epsilon-decay*epsilon
episode_lengths[i] = steps
print("\nDone in", steps, "steps".format(steps))

#Test con la política óptima
profit=[0]*100
for i in range(100):
state, reward, done = env.reset()
steps=0
step_reward=[0.0]*13
actions=[0]*13
R=0
while not done:

steps+= 1
action_probabilities = env.policyFunction(qtable1+qtable2,0)
action = np.random.choice(np.arange(len(action_probabilities)),
 p = action_probabilities)
actions[steps]=action
# take action
next_state, reward, done = env.step(action)
env.posX=next_state

# Update statistics
step_reward[steps] = reward
R=R+reward
# update state
state = next_state
profit[i]=R

print(actions)
print(profit)

sum=0
for i in range(100):
sum=sum+profit[i]
mean=sum/100
dt=0
for i in range(100):
dt=(mean-profit[i])**2+dt
dtf=np.sqrt(dt/100)
print(mean)
print(dtf)
```

Algoritmo Expected SARSA

```
env = Env()
```

```
epsilon = 0.8
total_episodes = 35000
decay_rate =epsilon/total_episodes
lr_rate = 0.5
gamma = 1
Qe = np.zeros((13,env.stateCount, env.actionCount))
episode_rewards=[0]*total_episodes
def choose_action(state):
action=0
if np.random.uniform(0, 1) < epsilon:
action = env.randomAction()
else:
action = np.argmax(Qe[state[1],state[0], :])
return action
#la estimación de los valores tiene en cuenta todas las posibles opciones
#a la hora de elegir action2
def learn(state, state2, reward, action, action2):
predict = Qe[state[1],state[0], action]
Q_esp=0
for i in range(env.actionCount):
Q_esp = Q_esp + Qe[state2[1],state2[0], i] * action2_probabilities[i]
target = reward +  Q_esp

Qe[state[1],state[0], action] = Qe[state[1],state[0], action]
 + lr_rate * (target-predict)
return Qe
# Start
rewards=0
for episode in range(total_episodes):
t = 0
state,reward,done =env.reset()
#choose action
action_probabilities = env.policyFunction(Qe,epsilon)
action = np.random.choice(np.arange(len(action_probabilities)),
 p = action_probabilities)
while not done:

nextstate, reward, done = env.step(action)
env.posX=nextstate
action2_probabilities = env.policyFunction(Qe,epsilon)
action2 = np.random.choice(np.arange(len(action2_probabilities)),
 p = action2_probabilities)
learn(state, nextstate, reward, action, action2)
state = nextstate
action = action2
t += 1
rewards+=1
episode_rewards[episode] += reward

epsilon=epsilon-decay_rate*epsilon

print ("Número de pasos: ", rewards/total_episodes)

#Test con la política óptima
profit=[0]*100
for i in range(100):
state, reward, done = env.reset()
steps=0
```

```
step_reward=[0.0]*13
actions=[0]*13
R=0
while not done:
steps+= 1
action_probabilities = env.policyFunction(Qe,0)
action = np.random.choice(np.arange(len(action_probabilities)),
 p = action_probabilities)
actions[steps]=action
# take action
next_state, reward, done = env.step(action)
env.posX=next_state

# Update statistics
step_reward[steps] = reward
R=R+reward
# update state
state = next_state
profit[i]=R
print(actions)
print(profit)

sum=0
for i in range(100):
sum=sum+profit[i]
mean=sum/100
dt=0
for i in range(100):
dt=(mean-profit[i])**2+dt
dtf=np.sqrt(dt/100)
print(mean)
print(dtf)
```

Aprendizaje multi-agente

```
#creacion del entorno

#nuevas acciones-precios
dicactions={0:725,1:775,2:825,3:875,4:925,5:975}
n_piezas=105  #numero de piezas en stock
n_periodos=12 #numero de meses para vender las piezas
#matrices de demanda
R1 = np.array([range(10,16),range(9,15),range(8,14),range(7,13),
range(6,12),range(5,11)])
R2 = np.array([range(10,4,-1),range(11,5,-1),range(12,6,-1),
range(13,7,-1),range(14,8,-1),range(15,9,-1)])
class Env2():
def __init__(self):
self.size = n_piezas+1;
self.periods=n_periodos
self.posX = [n_piezas,n_periodos];
self.endX = 0;
self.actions = [0,1,2,3,4,5];
self.stateCount = self.size;
self.actionCount = len(self.actions);
```

```
def reset(self):
self.posX = [n_piezas,n_periodos];
self.done = False;
return [n_piezas,n_periodos],0, False;

# take action
def step(self,action1,action2):
nextState = [self.posX[0]-R1[action1][action2],self.posX[1]-1];

done = nextState[1]==self.endX;

reward = (dicactions[action1])*(self.posX[0]-nextState[0])-h*self.posX[0];


return nextState, reward, done;

# return a random action
def randomAction(self):
return np.random.choice(self.actions);

def policyFunction(self,qtable,epsilon):
Action_probabilities = np.ones(self.actionCount, dtype = float) *
 epsilon / self.actionCount
best_action = np.argmax(qtable[self.posX[1]][self.posX[0]])
Action_probabilities[best_action] += (1.0 - epsilon)
return Action_probabilities


#OPPONENT MODELLING

env = Env2()
qtable = np.zeros([n_periodos+1,env.stateCount, env.actionCount,
 env.actionCount]).tolist()
Otable = np.zeros([n_periodos+1,env.stateCount, env.actionCount]).tolist()
Ns = np.ones([n_periodos+1]).tolist()
Nsa = np.ones([n_periodos+1, env.actionCount]).tolist()
# hyperparameters
epochs = 5000
epsilon = 0.8
decay = epsilon/epochs
#stats
episode_lengths=[0]*epochs
episode_rewards=[0]*epochs

for i in range(epochs):
state, reward, done = env.reset()
steps=0
while not done:

steps+= 1
action_probabilities = env.policyFunction(Otable,epsilon)
action = np.random.choice(np.arange(len(action_probabilities)),
 p = action_probabilities)
#opponent action
if env.posX[1]>9:
action0=random.randrange(4,6,1)
elif env.posX[1]>6:
```

```
action0=random.randrange(3,5,1)
elif env.posX[1]>3:
action0=random.randrange(2,5,1)
else:
action0=random.randrange(0,4,1)
# take action
next_state, reward, done = env.step(action,action0)
env.posX=next_state

# Update statistics
episode_rewards[i] += reward
# Update Otable
X=list(map(lambda x: x / Ns[state[1]], Nsa[state[1]] ))
Y=qtable[state[1]][state[0]][action]
Otable[state[1]][state[0]][action]=sum(i[0] * i[1] for i in zip(X, Y))
# update qtable value with Bellman equation
qtable[state[1]][state[0]][action][action0] = reward +
  max(Otable[state[1]-1][next_state[0]])
# update N
Nsa[state[1]][action0]=Nsa[state[1]][action0]+1
Ns[state[1]]= Ns[state[1]]+1
# update state
state = next_state
# The more we learn, the less we take random actions
epsilon = epsilon-decay*epsilon
episode_lengths[i] = steps
print("epoch #", i+1, "/", epochs)



#test con política óptima
state, reward, done = env.reset()
steps=0
step_reward=[0]*13
actions=[0]*13
while not done:
# count steps to finish game
steps+= 1
action_probabilities = env.policyFunction(Otable,0)
action = np.random.choice(np.arange(len(action_probabilities)),
 p = action_probabilities)
actions[steps]=action
# take action
if env.posX[1]>9:
action0=random.randrange(4,6,1)
elif env.posX[1]>6:
action0=random.randrange(3,5,1)
elif env.posX[1]>3:
action0=random.randrange(2,5,1)
else:
action0=random.randrange(0,4,1)
next_state, reward, done = env.step(action,action0)
env.posX=next_state

# Update statistics
step_reward[steps] = reward
# update state
state = next_state
```

```
episode_length = steps

print("\nDone in", steps, "steps".format(steps))
print(step_reward)
print(sum(step_reward))
a=dicactions.values()
b=list(a)
print(actions)


#entorno con dos agentes aprendiendo

#CREATE ENVIROMENT
import gym
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import random

dicactions={0:725,1:775,2:825,3:875,4:925,5:975}
n_piezas=210  #numero de piezas en stock
n_periodos=24 #numero de meses para vender las piezas
R1 = np.array([range(10,16),range(9,15),range(8,14),range(7,13),
range(6,12),range(5,11)])
R2 = np.array([range(10,4,-1),range(11,5,-1),range(12,6,-1),
range(13,7,-1),range(14,8,-1),range(15,9,-1)])

class Env2():
def __init__(self):
self.size = n_piezas+1;
self.periods=n_periodos
self.posX = [n_piezas,n_piezas,n_periodos];
self.endX = 0;
self.actions = [0,1,2,3,4,5];
self.stateCount = self.size;
self.actionCount = len(self.actions);

def reset(self):
self.posX = [n_piezas,n_piezas,n_periodos];
self.done = False;
return [n_piezas,n_piezas,n_periodos],0,0, False;

# take action
def step(self,action1,action2):
nextState = [self.posX[0]-R1[action1][action2],self.posX[1]
-R2[action1][action2],self.posX[2]-1];

done = nextState[2]==self.endX;

reward = (dicactions[action1])*(self.posX[0]-nextState[0])-5*self.posX[0];
reward2 = (dicactions[action2])*(self.posX[1]-nextState[1])-5*self.posX[1];

return nextState, reward,reward2,done;

# return a random action
def randomAction(self):
return np.random.choice(self.actions);
```

```
def policyFunction(self,qtable,epsilon):
Action_probabilities = np.ones(self.actionCount, dtype = float) *
 epsilon / self.actionCount
best_action = np.argmax(qtable[self.posX[2]][self.posX[0]])
Action_probabilities[best_action] += (1.0 - epsilon)
return Action_probabilities
def policyFunction2(self,qtable,epsilon):
Action_probabilities = np.ones(self.actionCount, dtype = float) *
 epsilon / self.actionCount
best_action = np.argmax(qtable[self.posX[2]][self.posX[1]])
Action_probabilities[best_action] += (1.0 - epsilon)
return Action_probabilities



#DOBLE OPPONENT MODELLING

env = Env2()
qtable = np.zeros([n_periodos+1,env.stateCount, env.actionCount,
 env.actionCount]).tolist()
Otable = np.zeros([n_periodos+1,env.stateCount, env.actionCount]).tolist()
Ns = np.ones([n_periodos+1]).tolist()
Nsa = np.ones([n_periodos+1, env.actionCount]).tolist()
Otable2 = np.zeros([n_periodos+1,env.stateCount, env.actionCount]).tolist()
Ns2 = np.ones([n_periodos+1]).tolist()
Nsa2 = np.ones([n_periodos+1, env.actionCount]).tolist()
qtable2 = np.zeros([n_periodos+1,env.stateCount, env.actionCount, env.actionCount])
# hyperparameters
epochs = 1000
epsilon = 0.8
decay = epsilon/epochs
#stats
episode_rewards=[0]*epochs
episode_rewards2=[0]*epochs

for i in range(epochs):
state, reward, reward2, done = env.reset()
steps=0
while not done:

steps+= 1
action_probabilities = env.policyFunction(Otable,epsilon)
action = np.random.choice(np.arange(len(action_probabilities)),
 p = action_probabilities)
#oponent action
action_probabilities2 = env.policyFunction2(Otable2,epsilon)
action0 = np.random.choice(np.arange(len(action_probabilities2)),
 p = action_probabilities2)
# take action
next_state, reward,reward2, done = env.step(action,action0)
env.posX=next_state

# Update statistics
episode_rewards[i] += reward
episode_rewards2[i] += reward2
# Update Otable
X=list(map(lambda x: x / Ns[state[2]], Nsa[state[2]] ))
Y=qtable[state[2]][state[0]][action]
```

```
Otable[state[2]][state[0]][action]=sum(i[0] * i[1] for i in zip(X, Y))
# Update Otable2
X=list(map(lambda x: x / Ns2[state[2]], Nsa2[state[2]] ))
Y=qtable2[state[2]][state[1]][action0]
Otable2[state[2]][state[1]][action0]=sum(i[0] * i[1] for i in zip(X, Y))
# update qtable value with Bellman equation
qtable2[state[2]][state[1]][action][action0] = reward +
  max(Otable2[state[2]-1][next_state[1]])
qtable[state[2]][state[0]][action][action0] = reward2 +
 max(Otable[state[2]-1][next_state[0]])
# update N
Nsa[state[2]][action0]=Nsa[state[2]][action0]+1
Ns[state[2]]= Ns[state[2]]+1
Nsa2[state[2]][action]=Nsa[state[2]][action]+1
Ns2[state[2]]= Ns[state[2]]+1
# update state
state = next_state
# The more we learn, the less we take random actions
epsilon = epsilon-decay*epsilon
print("epoch #", i+1, "/", epochs)
```